UNIVERSIDAD CARLOS III DE MADRID

TRABAJO FIN DE GRADO

SISTEMA DE INYECCIÓN DE FALLOS PARA EVALUACIÓN DE LA FIABILIDAD EN FPGAs

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

Autor: Adrián Rodríguez Bodas

Tutor: Luis Alfonso Entrena Arrontes

Leganés, Septiembre de 2015

Agradecimientos

Me gustaría dedicar este espacio a todas aquellas personas que me han apoyado a lo largo de la carrera y en especial durante la realización de este trabajo.

En primer lugar, dar las gracias a mi familia. A mis padres por todo el apoyo y cariño que me han dado, por ayudar a levantarme en los momentos de dificultad y por enseñarme a no conformarme en los buenos y que el esfuerzo es la clave del éxito. A mi hermana, por ponerme unas metas tan altas y servirme de ejemplo.

A mis compañeros de clase, por estar siempre ahí para ayudar, por compartir tantos momentos buenos y otros no tanto.

A mis amigos de toda la vida, los Aborígenes, por ayudar a desconectar, por sacarme de casa y estar siempre ahí.

Y finalmente a Luis por ofrecerme la oportunidad de realizar este trabajo, por su ayuda y sus consejos.

Gracias a todos.

Resumen

En este trabajo se pretende desarrollar una aplicación que permita realizar campañas de inyección de fallos aleatorias en FPGAs. Para ello se utilizará un módulo IP de la empresa Xilinx, el Soft Error Mitigation (SEM), que será el encargado de inyectar fallos en la memoria de configuración de la FPGA. El control de la aplicación se realizará mediante un microprocesador embebido en una FPGA tipo Zynq.

En un primer lugar se desarrollarán los elementos hardware, incluyendo en el diseño el microprocesador y el SEM. La interconexión de los dos elementos se realizará mediante GPIO y el bus AXI. Para este último será necesario el desarrollo de un módulo para conectar el SEM al bus y así conseguir un correcto entendimiento mediante el protocolo AXI4-lite. En este proceso se utilizará un lenguaje de descripción de hardware (VHDL) junto al diseño con bloques IP.

Una vez realizado el diseño hardware, se pasará al desarrollo del software de control de la aplicación, programando el funcionamiento en lenguaje C. El diseño permitirá la inyección cíclica de múltiples errores pseudo-aleatorios y su detección/corrección en caso de que fuese posible. Al finalizar, emitirá un informe sobre los errores que se han producido y su posible corrección.

Por último, se probará el diseño con un circuito de prueba, el cual se replicará varias veces y comparará su salida indicando si ha sucedido un error o no.

Abstract

This project aims to develop an application that allows to perform random fault injection campaigns in FPGAs. To this purpose we will use an IP from the company Xilinx, the Soft Error Mitigation (SEM), which will be responsible for injecting faults in the configuration memory of the FPGA. The control of the application is performed by a microprocessor embedded in a Zynq FPGA.

Firstly, the hardware elements will be developed. The interconnection of the microprocessor and SEM core will be made by AXI bus and GPIO. To connect the SEM and AXI bus it will be necessary to develop a block for a correct understanding by AXI4-lite protocol. In this process, a hardware description language (VHDL) and IP blocks are used.

Then, we will continue with the software development of control of the application. This will be programed in C language. The design allows cyclic injection of multiple random errors and detection/correction if it is possible. When finished, a report on errors that have occurred and their possible correction is issued.

Finally, the design will be tested with a test circuit, which will be replicated several times and it will compare its output to know if an error has occurred.

Índice

Índice de Figuras

Índice de Tablas

Listado de Acrónimos

FPGA	Field Programmable Gate Array
SEM	Soft Error Mitigation
IP	Intellectual Properties
PSOC	Programmable System on Chip
PL	Programmable Logic
PS	Processing System
SEU	Single Event Upset
SEE	Single Event Effect
ASIC	Application Specific Integrated Circuit
AXI	Advanced Extensible Interface
AMBA	Advanced Microcontroller Bus Architecture
HDL	Hardware Description Language
HLS	High-Level Synthesis
GPIO	General Purpose Input/Output
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus
ICAP	Internal Configuration Access Port
PCAP	Processor Configuration Access Port
MIO	Multiplexed Input/Output
EMIO	Extended Multiplexed Input/Output
SDK	Software Development Kit
GIC	Generic Interrupt Controller

1.Introducción

1.1. Motivación

En aplicaciones electrónicas que requieren una alta fiabilidad es necesario evaluar el efecto producido por posibles fallos y comprobar que el circuito es suficientemente fiable. En concreto, desde el punto de vista de la microelectrónica uno de los principales problemas es el efecto que produce la radiación sobre los componentes, causándoles errores ya sean transitorios o permanentes. Este problema se vuelve más crítico aún en el sector espacial, debido a la inaccesibilidad de los componentes y al aumento de la radiación, ya que en la Tierra la atmosfera protege en parte ante estos efectos.

Con el desarrollo de la microelectrónica los componentes se han vuelto más resistentes ante los llamados efectos permanentes producidos por una exposición prolongada a la radiación, lo cual ha sido posible debido a la reducción del grosor del óxido de puerta y el aumento de las densidades del dopado, pero el desarrollo de los componentes ha propiciado un aumento de la sensibilidad de los circuitos ante efectos transitorios, los cuales pueden estar causados por partículas ionizantes (SEE) o por exposición a radiación γ.

Los SEE (Single Event Effects) se deben a partículas energéticas aisladas que impactan contra el circuito produciendo errores. Hay distintos tipos de SEE pudiendo ser destructivos y no destructivos, siendo el de interés principal en este proyecto los SEU (Single Event Upset) catalogados como no destructivos. Este evento produce una corrupción de la información contenida en una memoria o en la salida de *latches* y *flip-flops*. Las FPGAs (Field Programmable Gate Array) debido a la memoria de configuración son muy críticas ante este tipo de errores, ya que cualquier cambio producido por un SEU puede modificar el circuito programado.

El uso de FPGAs en aplicaciones espaciales se ha incrementado en los últimos años debido a su versatilidad y el menor coste que los ASICs. Por contrapartida, estas son más sensibles a los SEU. Por ello se han desarrollado nuevas herramientas para testar y mejorar la fiabilidad de los circuitos ante estos problemas.

1.2. Objetivos

El objetivo de este trabajo es implementar una aplicación que permita realizar campañas de inyección de fallos aleatorias y presentar los resultados al usuario. Para este propósito se utilizará el módulo SEM de Xilinx y para el control de la aplicación se utilizará un microprocesador.

La implementación se realizará con una FPGA tipo Zynq, la cual tiene un microprocesador embebido, desde el que se controlará el módulo encargado de la inyección y corrección de errores (SEM). Tras esto se validará la solución con un ejemplo de prueba.

1.3. Organización de la memoria

La memoria del trabajo se ha querido organizar de una forma clara y concisa, buscando la fácil comprensión del lector. Para ello se ha dividido en capítulos y subcapítulos:

Capítulo 1: Se realizará una breve introducción a los errores en componentes microelectrónicos y los objetivos de este trabajo.

Capítulo 2: Se describe la tecnología utilizada en el trabajo, FPGA tipo Zynq y módulo SEM. También se describen las herramientas de diseño de Xilinx utilizadas.

Capítulo 3: Contiene una descripción detallada de la solución. Además se analizan otras posibles soluciones tenidas en cuenta en un principio.

Capítulo 4: Se detalla la realización de una validación experimental con un circuito de prueba y se analizan sus resultados.

Capítulo 5: Conclusiones del trabajo.

Capítulo 6: Presupuesto para el desarrollo del trabajo.

2.Tecnología y herramientas para la inyección de fallos

En la actualidad el campo de la inyección de fallos en circuitos microelectrónicos está ampliamente cubierto por distintos dispositivos. En este capítulo se quiere presentar las herramientas utilizadas en este trabajo, incluyendo tanto los elementos hardware como las herramientas de diseño software.

2.1. Zynq

La placa utilizada será la Zedboard, una placa de desarrollo de bajo coste perteneciente a la familia de Xilinx Zynq-7000. Esta se caracteriza por contener un sistema PSoC (Programmable System on Chip) formado por dos partes diferenciadas e interconectadas: una parte no programable (PS, Processing System) y otra programable (PL, Programmable Logic). La arquitectura puede verse en la *FIGURA 1*. Para la comunicación entre ambas partes del sistema, se emplea el interfaz AXI (Advanced Extensible Interface), el cual es parte de la arquitectura AMBA (Advanced Microcontroller Bus Architecture) de ARM. Esta interfaz proporciona un rendimiento alto, con conexión punto a punto y canales distintos para lectura y escritura.

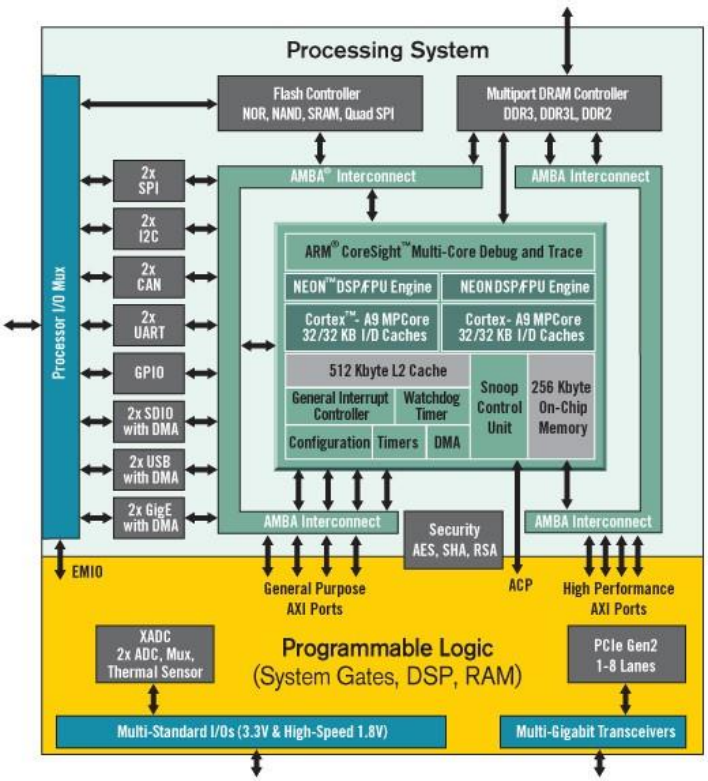

FIGURA 1: ARQUITECTURA ZYNQ-7000

La parte no programable (PS) la constituye un procesador ARM Cortex-A9 MPCore de doble núcleo junto a otros recursos, que asociados forman una APU (Application Processing Unit), además de un circuito generador de reloj, memoria caché e interfaces de memoria y periféricos. La arquitectura del PS puede verse en la *FIGURA 2*. Esta arquitectura soluciona el problema de espacio en FPGAs tradicionales al tener que incluir un sistema completo con sus procesadores, memorias, periféricos o interfaces. En las Zynq ya vienen incorporados estos recursos, dejando todo el espacio libre de la FPGA para ser implementado por el usuario.

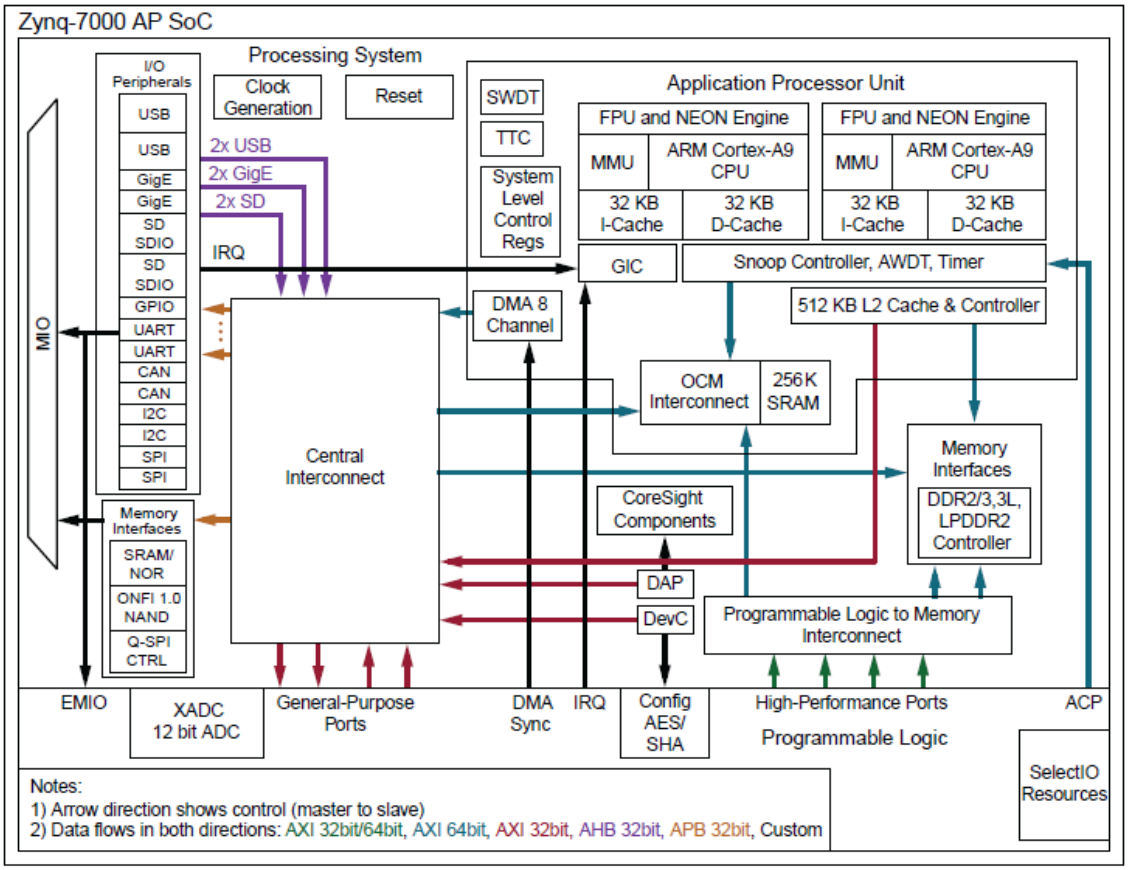

FIGURA 2: ZYNQ PROCESSING SYSTEM

La segunda parte principal de la arquitectura Zynq es la lógica programable (PL), basada en FPGAs de la serie 7 de Xilinx como Kintex-7 y Artix-7. Está formada por bloques lógicos programables (CLB), que junto a una matriz de interconexiones, forman una FPGA de uso general. Cada CLB a su vez está formada por dos *slices*, los cuales contienen recursos para implementar circuitos combinacionales o secuenciales, cada *slice* está formado por cuatro LUTs (Lookup Table) y por ocho *flip-flops*. Además hay circuitos aritméticos y bloques de entrada y salida para pines físicos.

La familia Zynq es un sistema basado en microprocesador, por lo que el arranque se efectúa siempre desde el PS, cargando la configuración desde la memoria ROM. El PL puede configurarse como parte del proceso de arranque o en un momento posterior, permitiendo también una reconfiguración parcial del PL.

La tarjeta Zedboard incluye además gran cantidad de periféricos que ofrecen gran versatilidad en el desarrollo de distintas aplicaciones. Algunos de ellos están conectados por defecto a ciertas partes de la arquitectura, aunque en general suelen ser configurables, permitiendo conectarlos a unos u otros pines. Algunos de estos periféricos están representados en la *FIGURA 3*:

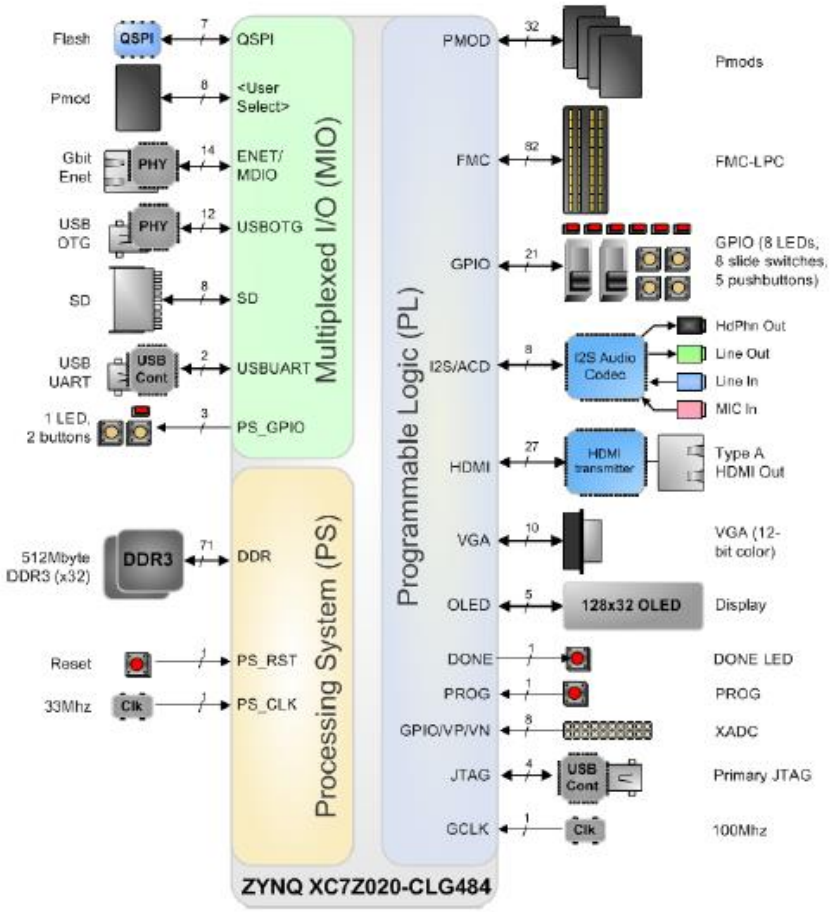

FIGURA 3: DIAGRAMA DE BLOQUES HARDWARE DE LA TARJETA ZEDBOARD

2.2. Herramientas de diseño de Xilinx

Con la aparición de la serie 7, Xilinx desarrolló un nuevo conjunto de herramientas de diseño para sustituir al anterior entorno de trabajo, ISE Design Suite, debido a las limitaciones que ofrecía este ante algunas nuevas funcionalidades de los productos de la serie 7. De esta manera surgió Vivado Design Suite, un entorno de trabajo SoC orientado al trabajo con IPs (Intellectual Properties) y al trabajo a nivel de sistema, es compatible con todos los dispositivos de las familias de la serie 7 de Xilinx y la familia Ultrascale. Vivado incorpora su propio simulador a diferencia de versiones anteriores de ISE, que trabajaban con ModelSim. Entre las herramientas que tiene Vivado se encuentra Vivado HLS que permite la creación de IPs mediante la descripción en lenguaje C, C++ y System C, lo que permite reducir el tiempo empleado en el desarrollo. La versión utilizada para este proyecto fue Vivado v2014.4. Entre las herramientas utilizadas se encuentran:

- **Vivado:** es el programa principal, encargado del diseño hardware. Permite tanto la configuración de PS de la plataforma Zynq como de PL. Permite una fácil integración e interconexión de bloques IP, además de programación con lenguajes de descripción de hardware como VHDL o Verilog. Vivado permite sintetizar diseños, realizar análisis de tiempo, examinar diagramas RTL o simular la reacción de un diseño ante distintos estímulos. Vivado ofrece integración con las demás herramientas de la Suite, facilitando su uso al usuario. Dispone de algunas ventajas sobre otras herramientas, que permiten automatizar procesos, reduciendo de esta manera el tiempo de desarrollo de un diseño.

- **Xilinx SDK:** es la herramienta encargada del desarrollo software ejecutable en una plataforma Zynq previamente desarrollada en Vivado. Contiene las librerías y drivers de los dispositivos utilizados, permitiendo además su configuración. También permite la programación de la FPGA y la generación de FSBLs (First Stage Boot Loader) para sistemas operativos.

2.3. SEM (Soft Error Mitigation)

Con el objetivo de aumentar la fiabilidad en sus FPGA, Xilinx proporciona un método para administrar los errores producidos por radiación ionizante (SEE). El controlador SEM es capaz de detectar y corregir los errores producidos en la memoria de configuración de la FPGA (SEU). El SEM también es capaz de inyectar errores simulando fallos producidos por radiación. Esta función puede servir para analizar la fiabilidad de una aplicación, comprobando la sensibilidad y vulnerabilidad ante una exposición a radiación.

El controlador SEM implementa cinco funciones principales: inicialización, detección de errores, inyección de errores, corrección de errores y clasificación de errores. Todas las funciones son opcionales, excepto inicialización y detección. Las funciones que se desee implementar han de ser seleccionadas en el proceso de configuración y generación del IP.

La comunicación del SEM con el resto de dispositivos se produce a través de interfaces, que según las funciones opcionales elegidas estarán disponibles o no, los interfaces y sus puertos pueden verse en la *FIGURA 4*:

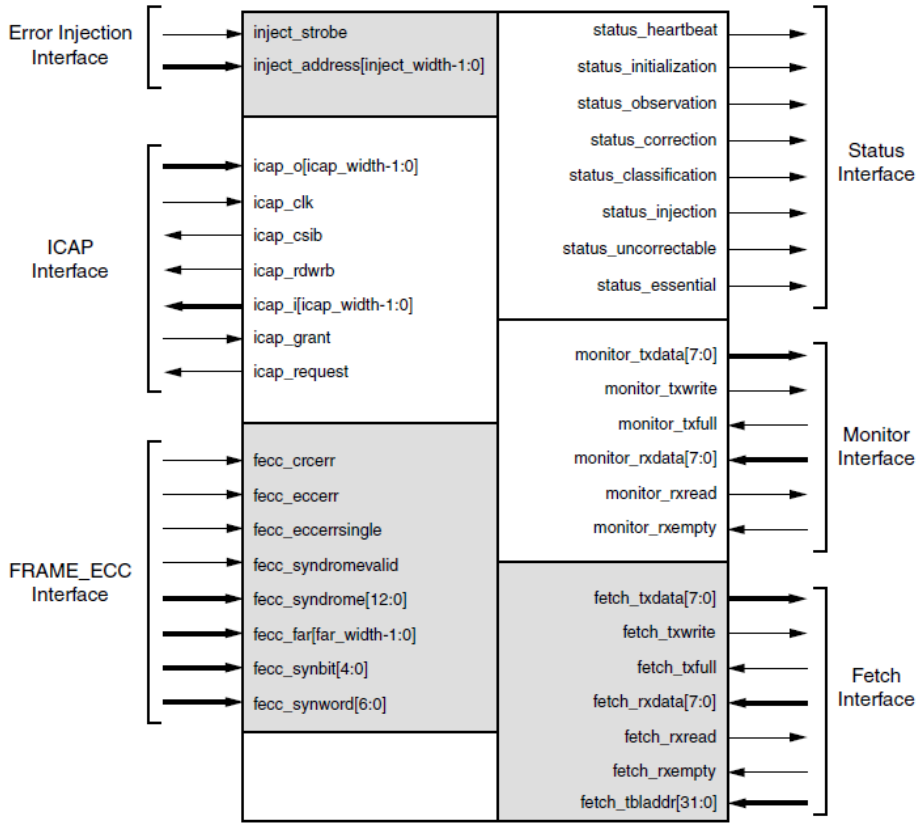

FIGURA 4: PUERTOS DEL CONTROLADOR SEM

- **ICAP Interface:** conexión punto a punto entre el SEM y la primitiva ICAP, la cual permite leer y escribir el acceso a los registros dentro del sistema de configuración de la FPGA.
- **FRAME_ECC Interface:** conexión punto a punto entre el controlador SEM y la primitiva FRAME_ECC. Los puertos de la primitiva son solo de salida, proporcionando al SEM una ventana para la detección de errores lógicos en el sistema de configuración de la FPGA.
- **Status Interface:** proporciona un conjunto de salidas que indican por nivel alto lo que el controlador SEM está haciendo.
- **Error Injection Interface:** está compuesto de un conjunto de entradas para ordenar al SEM la dirección y el momento de inyección de error. Solo es utilizado en caso de estar habilitada la inyección de errores.
- **Monitor Interface:** proporciona un mecanismo para interactuar con el usuario. Permite visualizar el estado en que se encuentra el controlador e información sobre los errores encontrados. También permite inyectar errores mediante comandos.
- **Fetch Interface:** mecanismo para que el controlador SEM solicite un dato a una fuente externa. Durante la corrección o la clasificación puede necesitar buscar datos de configuración. El SEM escribe un comando en binario para pedir el dato y la fuente externa tiene que ser capaz de devolverle el dato pedido. Al configurar el controlador SEM se puede elegir el tipo de reparación, este interface solo será utilizado en caso de seleccionar la opción de reemplazar, si se elige reparar el error no se utilizará.

El comportamiento del SEM es cíclico, pasando por varios estados en los que realiza distintas acciones. Estos estados dan una visión general a nivel de sistema del funcionamiento del controlador SEM. Los estados son:

- **Inicialización:** durante el proceso de configuración de la FPGA el controlador SEM se mantiene desactivado. Al finalizar ese proceso comienza la inicialización del SEM cuando la señal *icap_grant* esté activada. Durante este estado la señal *status_initialization* está activada.
- **Observación:** el SEM observa el sistema de configuración de la FPGA para la detección de errores. Si el controlador no detecta errores y recibe un comando valido, realizará la acción pertinente. Los únicos comandos validos son "status_report" y "enter_idle". El primero mandará al SEM que realice un informe de situación y el segundo provocará el cambio de estado a *idle.* Durante este estado la señal *status_observation* está activada.
- **Corrección:** el controlador SEM intenta corregir los errores. Al finalizar el proceso pasa al estado de clasificación; si no puede corregir el error activa la señal *status_uncorrectable*. Durante este estado la señal *status_correction* esta activada. El controlador siempre pasa por este estado aunque la corrección este desactivada.

- **Clasificación:** el controlador clasifica los errores según se hayan podido corregir o no. Si el error es incorregible, el controlador no vuelve al estado de observación y la FPGA debe ser reconfigurada. Aunque la clasificación no este activada, el controlador siempre pasa por este estado. Durante este estado la señal *status_clasification* está activada.
- **Idle:** el controlador se mantiene inactivo durante este estado, a la espera de un comando valido o inyectar un error. Los únicos comandos validos en este estado son "enter_observation" y "status_report". El primero provocará el paso a estado de observación y el segundo pedirá un informe de situación. Este estado se indica mediante la desactivación de las cinco señales de estado.
- **Inyección:** el SEM realiza las inyecciones de error en este estado. Durante este estado *status_injection* está activada.
- **Fatal Error:** el controlador entra en este estado debido a un error grave. La FPGA debe ser reconfigurada para salir de este estado.

La *FIGURA 5* describe el comportamiento del SEM mediante un diagrama de flujo:

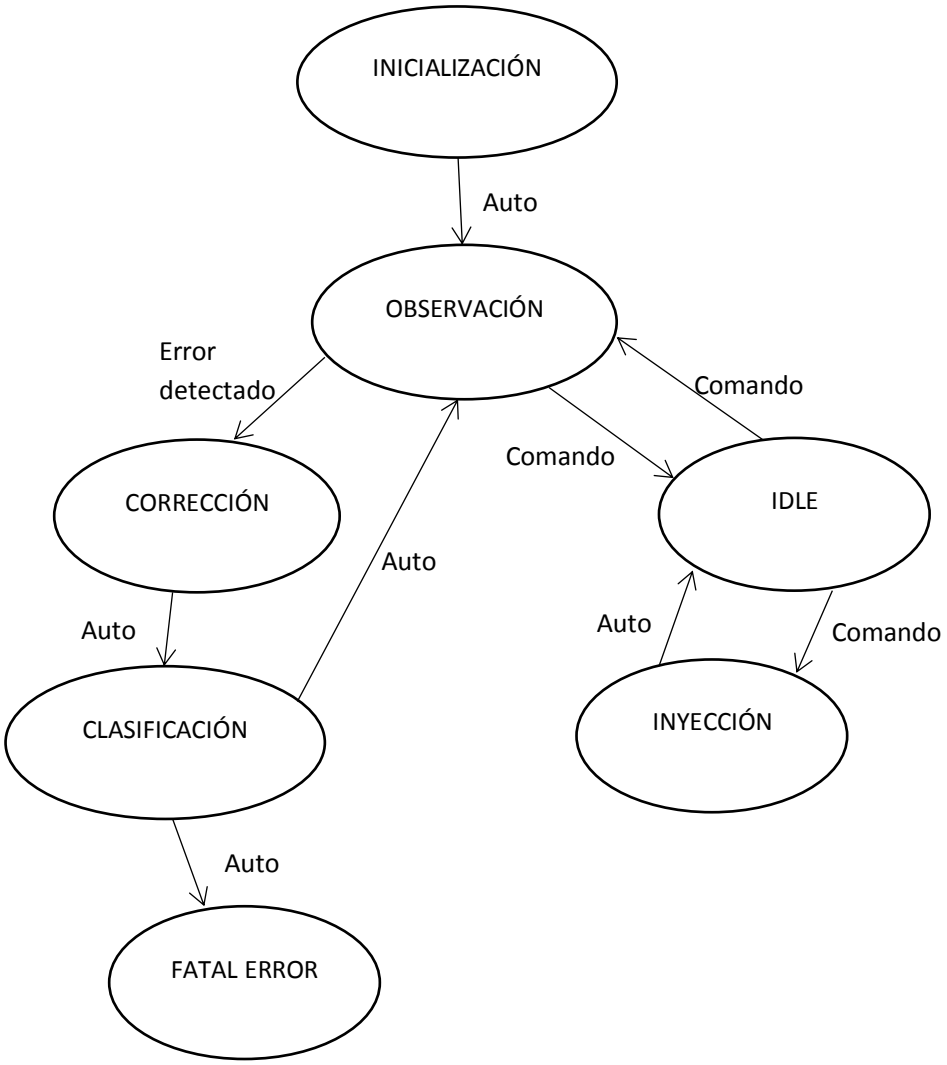

FIGURA 5: DIAGRAMA DE ESTADOS DEL SEM

Entre las opciones que permite el controlador SEM está elegir el tipo de corrección que se desea utilizar:

- Repair: es la reparación más simple, está basada en un algoritmo ECC (Error Correction Code).
- Enhanced Repair: reparación mejorada, solo disponible en la Serie 7 de Xilinx. Está basada en los algoritmos ECC y CRC (Cyclic Redundancy Check).
- Replace: este método reemplaza los bits defectuosos, recargando desde un soporte externo su información de configuración original. El *Fetch Interface* es necesario en este método de reparación.

3.Diseño de la aplicación

El módulo encargado de la inyección, detección y corrección de los fallos es el SEM, este IP es el centro de este proyecto. Todo el diseño a su alrededor va dirigido a conseguir su correcto funcionamiento, siempre controlado por el procesador de la Zynq. En este capítulo se describe el proceso de diseño de la aplicación, desde la creación y utilización de IPs, hasta la creación del bitstream y el software de la aplicación.

3.1. Análisis de posibles soluciones

En la mayoría de aplicaciones la solución no es única, ya que siempre podría utilizarse otra tecnología o podría plantearse el problema de distinta manera. El uso de cada tecnología tendrá unas ventajas e inconvenientes que el ingeniero deberá valorar para intentar tomar la decisión más óptima. Este proyecto no es distinto, y en su desarrollo se han encontrado múltiples opciones.

Las principales opciones surgieron en la interconexión del controlador SEM y el procesador (PS), utilizando distintas tecnologías para ello:

- Interconexión mediante GPIO (General Purpose Input/Output) : proporciona un diseño sencillo con una conexión punto a punto entre los dos principales módulos del diseño. El principal inconveniente es la cantidad de puertos GPIO que se necesitarían, un total de 51 GPIO. Para una aplicación de este tamaño un uso de tantos GPIOs no parece la solución más adecuada. Simplifica también el diseño software posterior ya que permite el trabajo con funciones de librería exclusivamente. Además pueden utilizarse los GPIO del procesador o un IP incluido en Vivado, AXI_GPIO que aumenta el número de puertos GPIO disponibles para el diseño. Esta solución se llegó a plantear y se comenzó su desarrollo, pero se abandonó por los inconvenientes anteriormente descritos. La solución planteada se describe brevemente en el *Apartado 3.2*.

- Interconexión mediante interfaz AXI: este diseño exige el desarrollo de un módulo para implementar el interfaz del protocolo AXI e internamente crear unos registros para seleccionar la señal o señales con las que interactuar mediante los canales de dirección de escritura y lectura. El diseño hardware y software es más complejo aunque la solución es más óptima, mandando todo el flujo de información a través del bus AXI.

- Interconexión mediante interfaz AXI y GPIO: esta es la solución elegida, la cual mezcla las dos anteriores, conectando por el bus AXI la señal *inject_address* de 40 bits y por los puertos GPIO las demás señales. Esta solución permite reducir considerablemente el número de puertos GPIO utilizados y mantener las ventajas de su uso en otros. Permite utilizar interrupciones con las señales de estado así poder crear un software más óptimo. Esta solución se describe en el *Apartado 3.3*.

3.2. Solución con interconexión GPIO

Para el diseño hardware se tienen dos bloques principales, el SEM y el PS. Cada uno de ellos hay que configurarlos adecuadamente y posteriormente desarrollar las interconexiones entre ellos.

Para configurar el SEM se creó un nuevo proyecto, en el que se incluyó el IP *Soft Error Mitigation* del catálogo de Vivado y utilizando el diseño de ejemplo que ofrece Xilinx, se completó el diseño del SEM. Para llegar a este punto se realizaron distintas configuraciones que se describirán posteriormente en el Capítulo 3.3.1. Una vez completado se creó un bloque IP que se exporta al proyecto principal para incluirlo en el diseño.

En la configuración del PS sí existen mayores diferencias respecto al diseño final. Esto es debido al uso de GPIO de un bloque IP del catálogo en vez de utilizar los GPIO del microprocesador, por lo que de los recursos del PS únicamente se utiliza la UART1 para la comunicación con el usuario configurada a 115200 baud y el Timer 0 para un uso posterior en el software a 111MHz. Además se habilita la conexión maestra del interfaz AXI para conectar los GPIO. Se habilita una fuente de reloj a 100MHz para los bloques interconectados mediante bus AXI y que también servirá como fuente de reloj para el SEM.

Debido al gran número de señales a conectar al GPIO, se decidió utilizar dos bloques IP iguales, uno con las conexiones de salida y otro con las de entrada. Este IP es configurable con lo que se asignaron las dimensiones adecuadas a cada uno, habilitando incluso el doble canal para el encargado de las señales de salida. Las configuraciones realizadas para los bloques AXI_GPIO pueden verse en la *FIGURA 6 y FIGURA 7*:

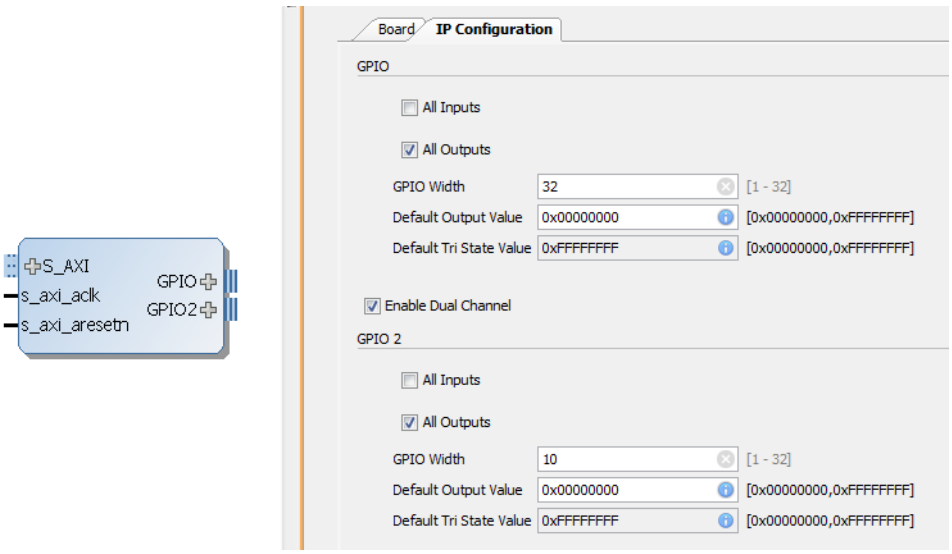

FIGURA 6: CONFIGURACIÓN DE AXI_GPIO OUTPUT

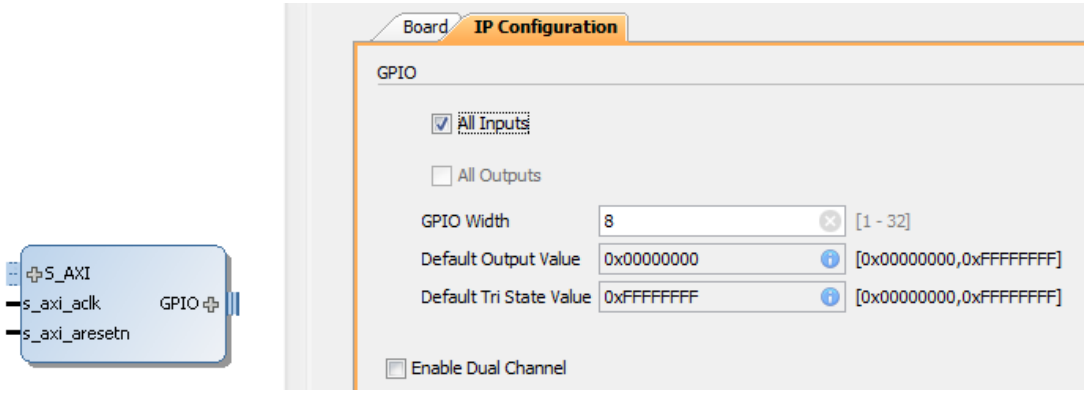

FIGURA 7: CONFIGURACIÓN DE AXI_GPIO INPUT

Estos bloques IP disponen de librerías con funciones específicas, lo que permite un desarrollo del software más sencillo, permitiendo interrupciones y un manejo similar a los GPIO del microprocesador. A su vez ofrecen un número de puertos mayor y configurable.

Una vez configurados todos los bloques, ya se pueden realizar todas las interconexiones. Los AXI_GPIO se conectan con el bloque AXI Interconnect que hace las funciones de maestro y este a su vez con el PS. Este bloque se utiliza para que un mismo maestro (PS) pueda controlar varios esclavos sin necesidad de habilitar más de una salida maestra para el interfaz AXI. A su vez Processor System Reset forma una fuente de reset para los bloques esclavos y para AXI Interconnect. Las conexiones de las señales del SEM con los GPIO se realizan programándolas posteriormente en VHDL en el archivo *HDL Wrapper*. La *FIGURA 8* presenta el resultado final:

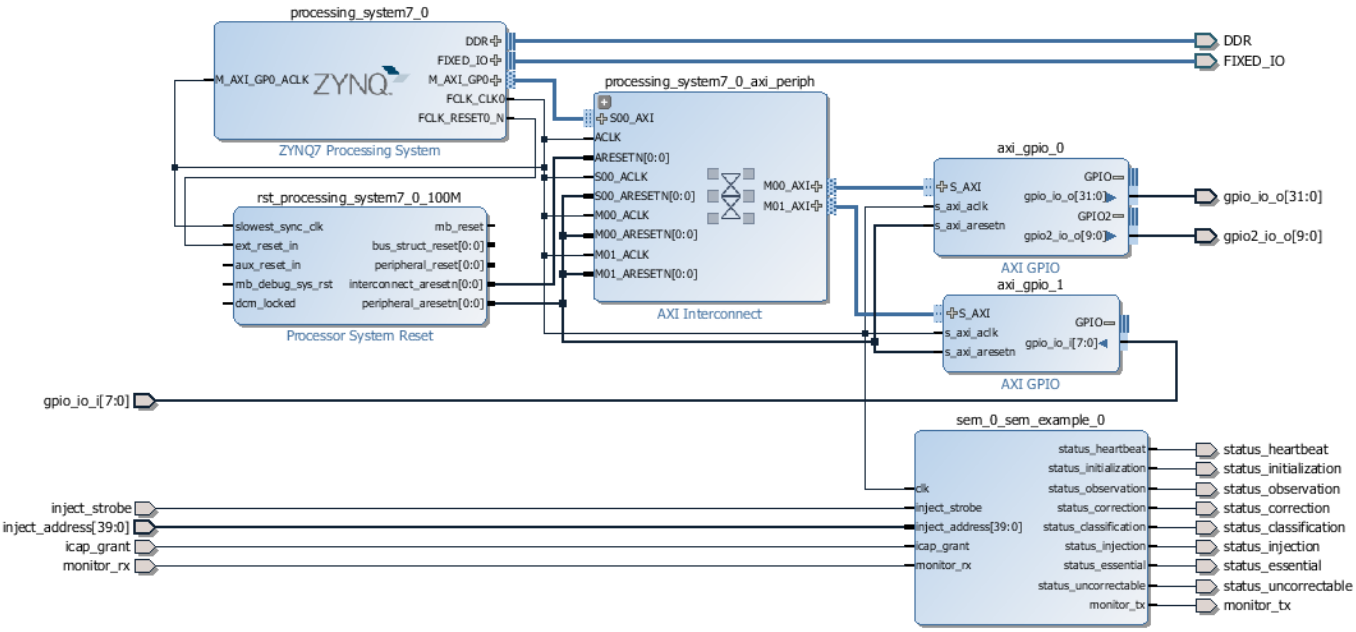

FIGURA 8: DISEÑO HARDWARE CON INTERCONEXIÓN GPIO

Como ya se ha comentado, esta propuesta fue abandonada a favor de la utilización del bus AXI. No obstante, este primer desarrollo sirvió como aprendizaje de los métodos y mecanismos de diseño en Vivado. También permitió apreciar los primeros inconvenientes en el diseño y ayudar a obtener una solución posterior más adecuada.

3.3. Solución con interconexión GPIO e Interfaz AXI

La solución hardware consta de dos partes principales, el controlador SEM y el procesador o PS, las cuales tienen que ser interconectadas y para ello podrían usarse distintos métodos. La solución elegida que se ha desarrollado consiste en interconexión mediante bus AXI y GPIO. Además del SEM y el procesador, es necesario un circuito independiente al que someter a la inyección de fallos, este circuito es implementado en la FPGA o PL y puede ser cualquier circuito digital. La estructura a desarrollar es la siguiente:

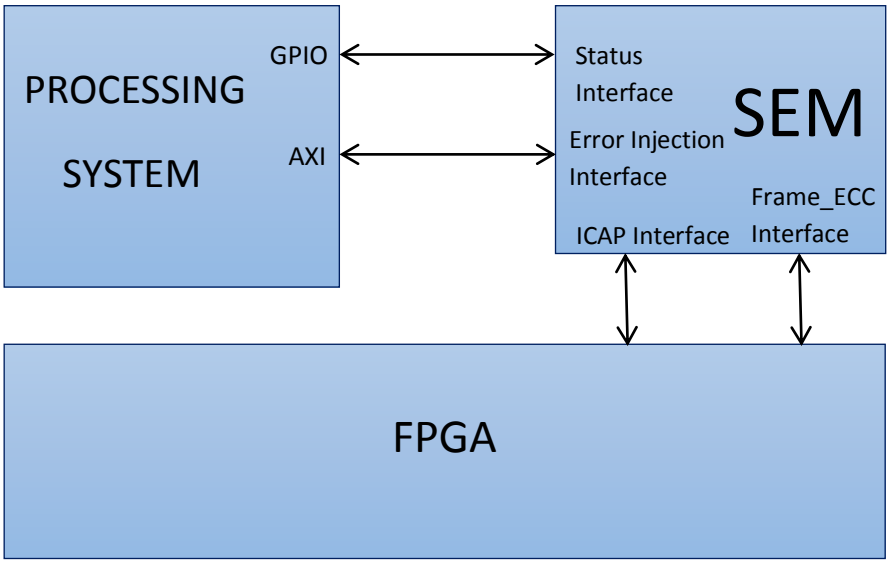

FIGURA 9: ESTRUCTURA DE LA APLICACIÓN

Como puede verse en la *FIGURA 9*, se conectan las señales del *Status Interface* al GPIO, de esta manera se podrán aprovechar las interrupciones para la aplicación software. El *Error Injection Interface* se conecta al bus AXI. Esto es debido a las dimensiones de la señal *Inject_address* (40 bits). Como el bus AXI es de 32 bits, hay que mandar la dirección para inyectar el error en dos partes. *ICAP Interface* y *Frame_ECC Interface* se conectan a sus respectivas primitivas y a través de ellas verán la FPGA.

El controlador SEM dispone de otros dos interfaces que no se han utilizado. El *Fetch Interface* no se utiliza debido a que no se elige el tipo de reparación de reemplazar y el *Monitor Interface*, aunque sí está implementado como una UART, no se utilizará debido a que la placa Zedboard solo dispone de un conector USB-UART y este tiene otro propósito en la aplicación.

3.3.1. Configuración del controlador SEM

Para configurar el SEM, en primer lugar se creó un nuevo proyecto con el objetivo de crear un bloque IP que poder añadir en el proyecto principal. Además hay que implementar un mecanismo para que la dirección de inyección de error se transmita a través del bus AXI. Por claridad y simplicidad se decidió crear otro bloque IP, que hiciera de puente entre el PS y el SEM.

En primer lugar, se añade al proyecto el IP *Soft Error Mitigation* desde el catálogo de Vivado. En este punto podemos elegir ciertas funciones del SEM.

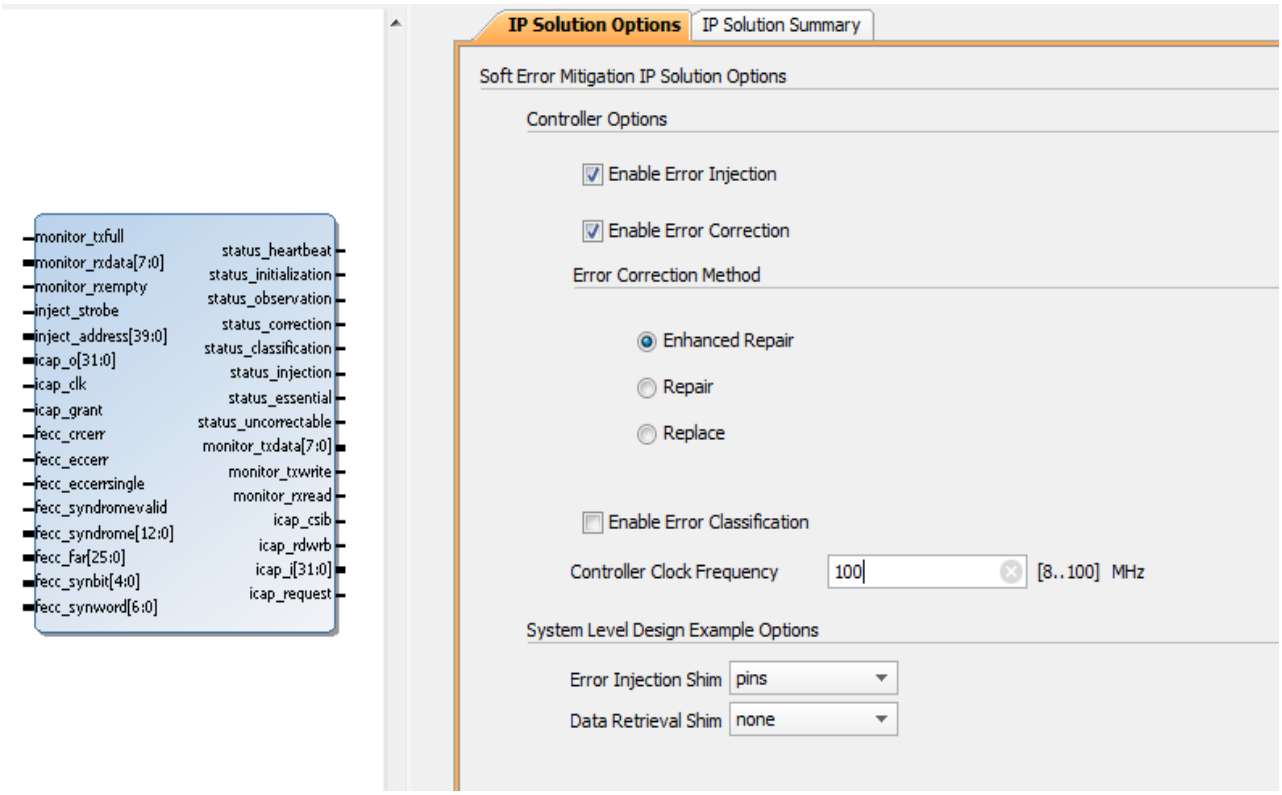

FIGURA 10: CONFIGURACIÓN SEM IP

Como se puede ver en la *FIGURA 10*, están habilitadas las funciones de inyección y corrección. La clasificación esta deshabilitada debido a que el Monitor Interface no será usado y por ello no interesa esta función. En cuanto al método de reparación, se eligió *Enhanced Repair* (Reparación mejorada). Este es más completo que la reparación normal, ya que está basado en los algoritmos ECC y CRC. La opción de reemplazo de los bits defectuosos se descarta, debido a que entonces habría que utilizar el *Fetch Interface* y elementos de memoria externos. La frecuencia del reloj se fija a 100MHz Esta es la frecuencia de la fuente de reloj del procesador para el PL (FCLK_CLK0). Esta frecuencia se selecciona en la configuración el PS posteriormente.

Junto al IP del controlador SEM, hay que incluir un diseño que permita su correcto funcionamiento. para ello se incluye el ejemplo de diseño para el SEM proporcionado por Xilinx.

A este diseño de ejemplo se le realizaron algunas modificaciones: en primer lugar la señal *icap_grant* se conectó a un puerto de entrada. Se tomó esta decisión debido a una recomendación indicada en el manual del controlador SEM, que recomendaba conectar esta señal a un GPIO cuando se utilizase el SEM en placas de la familia Zynq ya que si el interfaz ICAP intentaba acceder antes de que la ruta de acceso estuviera disponible podría provocar un funcionamiento inesperado en el controlador SEM. Durante el proceso de configuración de la FPGA, el procesador accede al PL a través de PCAP (Processor Configuration Access Port). Este puerto y el ICAP (Internal Configuration Access Port) están controlados por un mismo multiplexor, que impide el acceso de ambos a la vez. La misma situación se produce entre la salida del multiplexor y el JTAG, por lo que es necesario conectar *icap_grant* a un GPIO y posteriormente durante el diseño software, realizar un correcto manejo de los bits del registro de control del *Device Configuration Interface.*

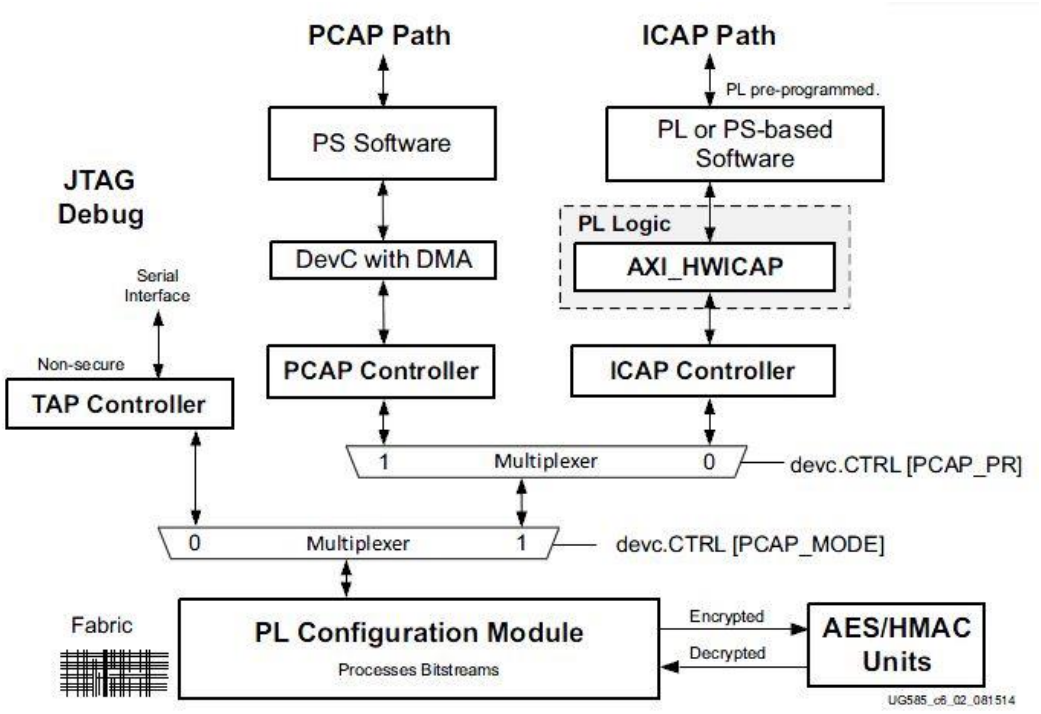

FIGURA 11: CONFIGURACIÓN DE ACCESO AL PL

Otra modificación realizada al diseño de ejemplo consistió en incluir un mecanismo que permitiera leer a tiempo las señales de estado *status_correction* y *status_classification*, ya que por el poco tiempo que se mantenían activas, el procesador no era capaz de llegar a leer su estado de activación a tiempo al saltar la interrupción del GPIO. Este consiste en un registro que permite tomar el valor '1' de la señal de estado pero no cambia al desactivarse dicha señal, únicamente vuelve a '0' al activarse la señal *reset_signal_status*. El código implementado es como el de la *FIGURA 12*, modificado para cada señal.

```
PROCESS(clk_ibufg)
BEGIN
IF clk_ibufg'EVENT AND clk_ibufg='1' THEN
    IF status_correction_internal='1' THEN
        status_correction<=status_correction_internal;
    ELSE
        IF reset_signal_status='1' THEN
            status_correction<='0';
        END IF;
    END IF;
END IF;
END PROCESS;
```

FIGURA 12: CÓDIGO REGISTRO SEÑALES DE ESTADO

Una vez realizadas las modificaciones se creó un nuevo IP, el cual se incluirá en el diseño final. El código VHDL esta adjunto en el *APENDICE 1.1*. La *FIGURA 13* representa el IP definitivo, con sus entradas y salidas.

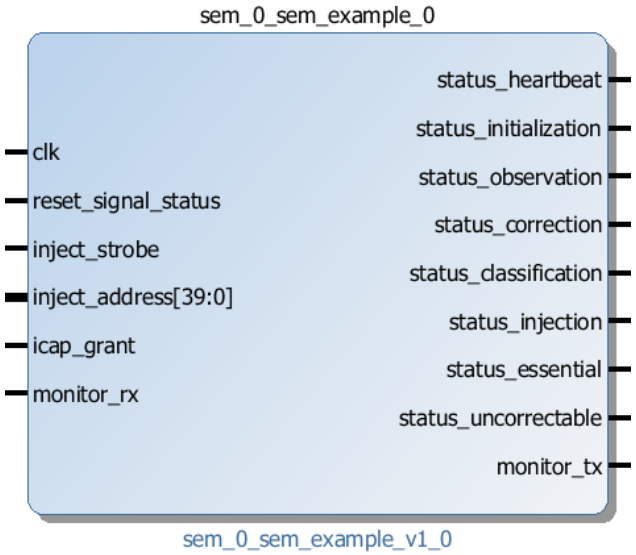

FIGURA 13: IP SOFT ERROR MITIGATION

Como se comentó, hay un bloque que hace de nexo entre el PS y el SEM. Este bloque está diseñado para hacer de puente entre la señal *inject_address* de 40 bits y el bus AXI de 32 bits, así que la información debe mandarse en dos fragmentos de 20 bits cada uno. Según el valor del bit 31 del dato mandado, los 20 bits inferiores del dato se escribirán en la parte superior o inferior de *inject_address*, si el bit 31 es '0' se escribirá en la parte inferior y si es '1' en la superior.

Para conseguir un correcto entendimiento, hay que diseñar un bloque que entienda el protocolo AXI4-lite del bus. Este protocolo se basa en una relación maestro-esclavo con señales de "Ready" y "Valid" en cada canal. Las señales del protocolo AXI4-lite están divididas en cinco canales además del reloj y el reset.

```
-- Clock and Reset
S_AXI_ACLK        : in std_logic;
S_AXI_ARESETN     : in std_logic;
-- Write Address Channel
S_AXI_AWADDR      : in  std_logic_vector(31 downto 0);
S_AXI_AWVALID     : in  std_logic;
S_AXI_AWREADY     : out std_logic;
-- Write Data Channel
S_AXI_WDATA       : in  std_logic_vector(31 downto 0);
S_AXI_WSTRB       : in  std_logic_vector(3 downto 0);
S_AXI_WVALID      : in  std_logic;
S_AXI_WREADY      : out std_logic;
-- Read Address Channel
S_AXI_ARADDR      : in  std_logic_vector(31 downto 0);
S_AXI_ARVALID     : in  std_logic;
S_AXI_ARREADY     : out std_logic;
-- Read Data Channel
S_AXI_RDATA       : out std_logic_vector(31 downto 0);
S_AXI_RRESP       : out std_logic_vector(1 downto 0);
S_AXI_RVALID      : out std_logic;
S_AXI_RREADY      : in  std_logic;
-- Write Response Channel
S_AXI_BRESP       : out std_logic_vector(1 downto 0);
S_AXI_BVALID      : out std_logic;
S_AXI_BREADY      : in  std_logic;
```

FIGURA 14: SEÑALES PROTOCOLO AXI4_LITE

Una vez implementado el periférico (bloque IP), se le asigna un rango de direcciones de memoria para escribir o leer sus registros, en este sentido los canales de dirección de lectura o escritura se encargan de dentro de ese rango seleccionar el registro en que debe escribirse o leerse el dato.

El canal de lectura del dato, toma el dato del registro indicado a través del canal de dirección de lectura y lo manda al maestro, que sería el PS.

El canal de escritura de datos, escribe el dato en el registro indicado por el canal de dirección de escritura y el canal de respuesta manda una señal codificada en 2 bits para indicar la correcta escritura o para indicar un fallo.

Para comprobar el correcto funcionamiento del bloque cumpliendo el protocolo, se realizó una simulación. En ella se aprecia cómo el manejo de las señales "Ready" y "Valid" es correcto y como se asigna el valor de la parte superior o inferior de *inject_address* según el bit 31 del dato de escritura.

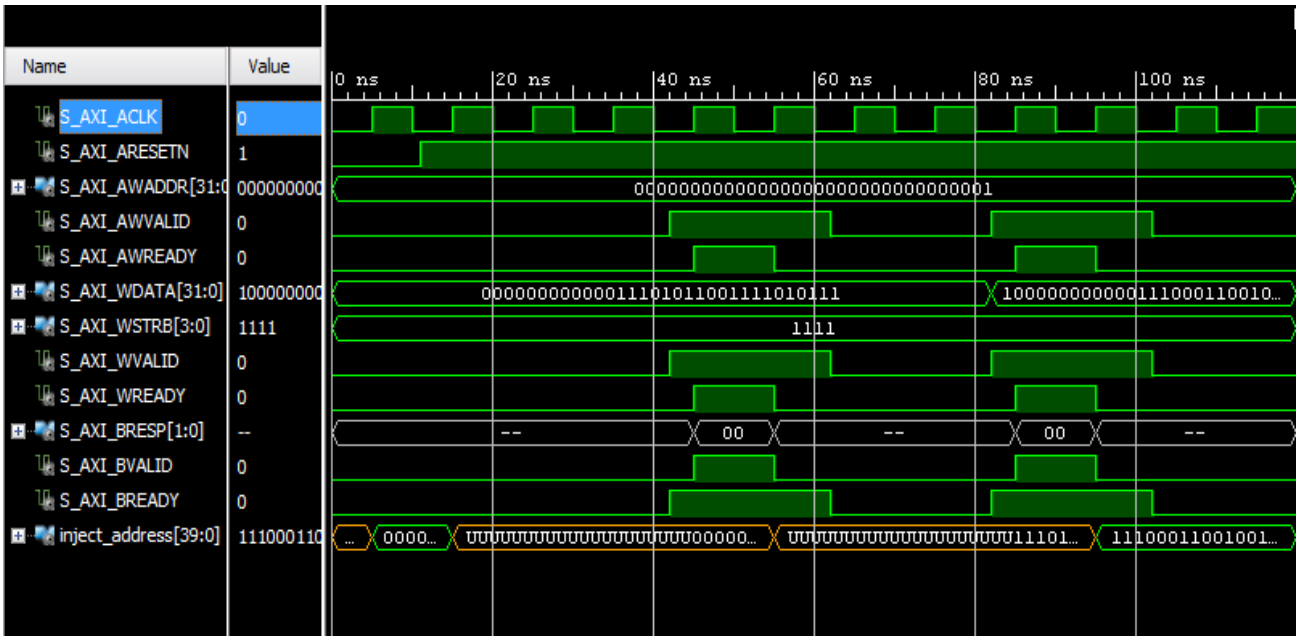

FIGURA 15: SIMULACIÓN PROTOCOLO AXI4-LITE

Una vez comprobado el funcionamiento se creó el bloque IP para incluir en el diseño, quedando un bloque como el de la *FIGURA 16*. En el *APENDICE 1.2*. está el código VHDL del diseño para interconectar el bus AXI y el controlador SEM.

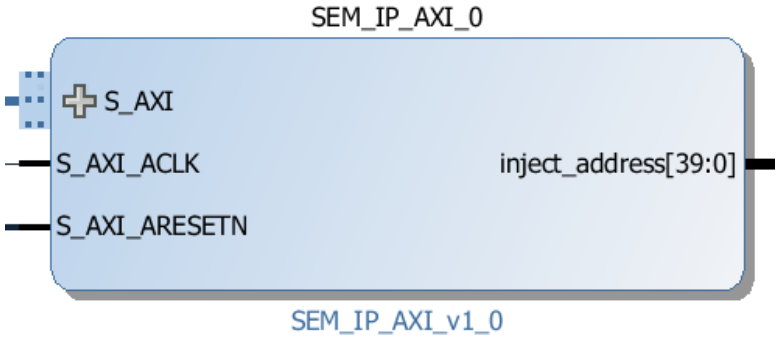

FIGURA 16: BLOQUE IP ESCLAVO AXI

3.3.2. Configuración del PS

Una vez configurado el controlador SEM, se puede comenzar a construir el sistema completo. Para ello una parte fundamental es el sistema de procesamiento (PS), el cual hay que configurar con distintas opciones que ofrece el IP incluido en el catálogo de Vivado.

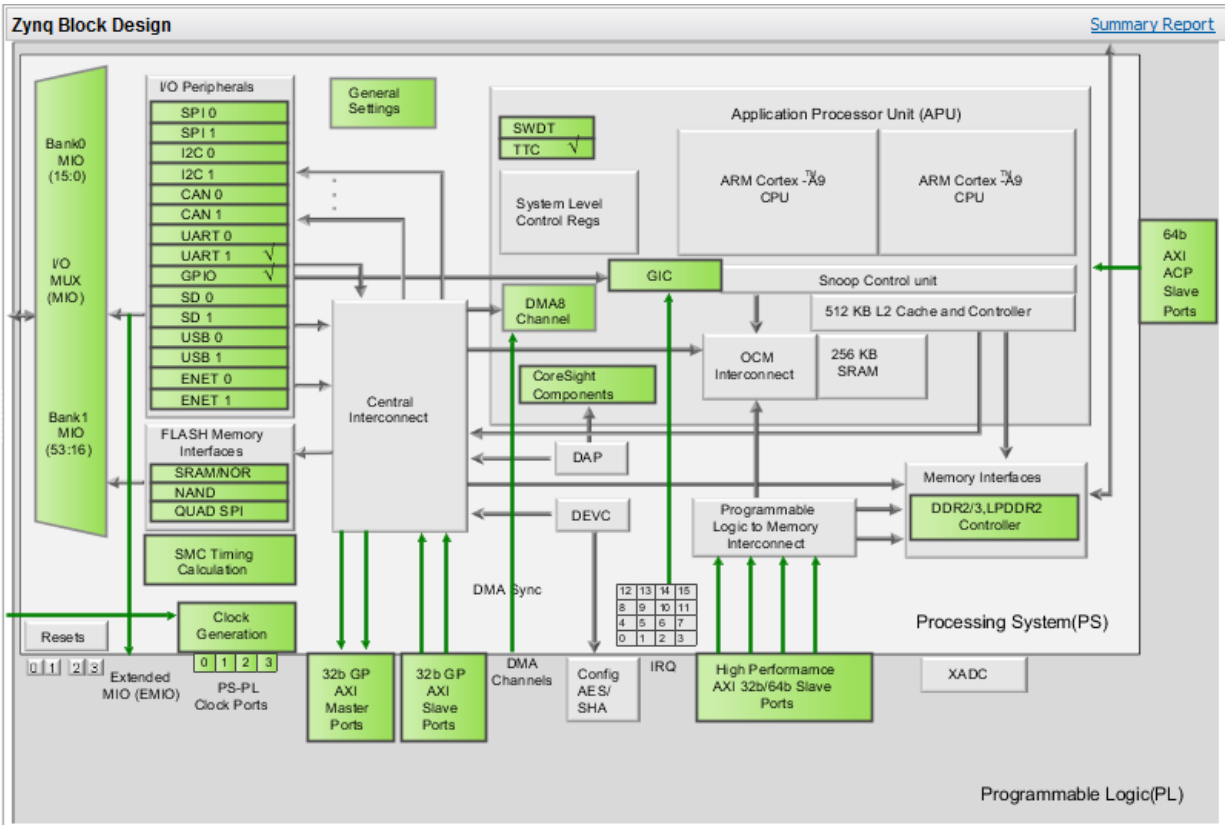

FIGURA 17: ZYNQ BLOCK DESIGN

El PS de Zynq está formado por distintos bloques que ofrecen gran versatilidad, dependiendo la aplicación. Algunos de ellos pueden estar habilitados o no, además de poder caracterizarlos dependiendo de las necesidades. La *FIGURA 17* muestra todos los bloques que forman el PS. Aquellos que están en verde pueden ser caracterizables y habilitados o no.

En primer lugar se van a seleccionar los periféricos de entrada y salida que son necesarios para el diseño, habilitándolos y deshabilitando los restantes. Se utilizará una UART conectada al puerto USB-UART de la placa Zedboard para la comunicación entre el usuario y el sistema. Para ello se habilita la UART 1, la cual ofrece un diseño más sencillo ya que por defecto ya está conectada al puerto de la placa, y se le asignó una velocidad de transmisión de 115200 baud. Se asignó a los puertos 48 y 49 del MIO (Multiplexed Input/Output), siendo el 48 el encargado de la transmisión (tx) y el 49 de la recepción (rx).

También es necesaria la utilización de GPIOs para la interconexión con el controlador SEM. Se configuraron a través del EMIO (Extended Multiplexed Input/Output), el cual permite además definir el ancho, fijado en 12.

Se habilitó el Timer 0, que será utilizado para el diseño software y se fijó su frecuencia en 111 MHz. También se configuró la frecuencia de la fuente de reloj para el SEM a 100MHz. La FPGA (PL) también usa esta fuente de reloj, aunque si fuese necesario podría activarse otra fuente de reloj y ajustar su frecuencia.

Por otra parte se habilitó un interfaz maestro para protocolo AXI al que poder conectar el SEM. Este se conectó a un módulo de interconexión AXI permitiendo conectar más de un periférico esclavo a un mismo maestro.

Otras configuraciones como el interfaz de memoria y sus características se mantuvieron con la configuración por defecto ya que su configuración de una u otra manera no interfería decisivamente en el diseño. Las demás características configurables del PS se deshabilitaron ya que no se usan.

El resultado es un bloque como el de la *FIGURA 18*, el cual dispone de las fuentes de reloj y reset para el PL, el interfaz maestro para protocolo AXI, el puerto GPIO y junto a estos aparecen por defecto los interfaces DDR y FIXED_IO. El primero es el interfaz de memoria del procesador y el segundo corresponde a las señales de entrada y salida que van conectadas a pines en la placa Zedboard.

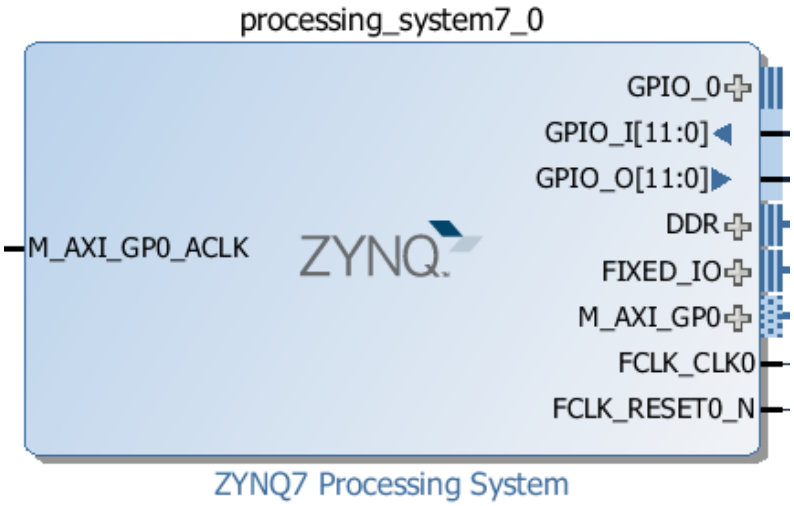

FIGURA 18: PROCESSING SYSTEM IP

3.3.3. Integración de elementos hardware

Con ambos bloques configurados se puede comenzar la integración de los elementos hardware para el diseño del sistema de inyección de fallos. Los bloques ya están preparados y únicamente falta la unión de los interfaces y señales correspondientes.

Junto a los bloques IP conocidos PS y SEM, hay que añadir bloques incluidos en el catálogo que funcionan como complemento necesario al utilizar el interfaz AXI para la interconexión.

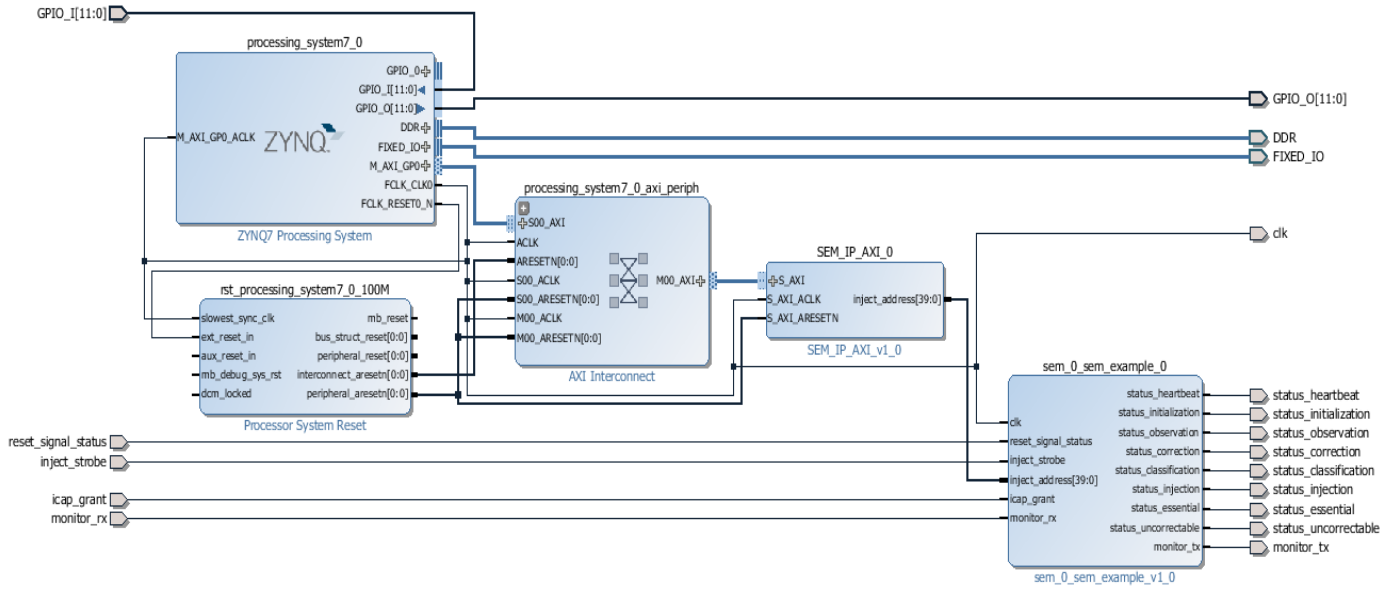

FIGURA 19: DISEÑO HARDWARE FINAL

En la *FIGURA 19* se pueden apreciar los bloques *AXI Interconnect* y *Processor System Reset* que son los bloques extra comentados. Estos bloques los añade Vivado automáticamente y cumplen las siguientes funciones:

- *AXI Interconnect* funciona como unión entre maestro y esclavos en el interfaz AXI. Permite controlar más de un esclavo desde un único maestro compartiendo el reloj y el reset. Permite la interconexión de hasta 64 esclavos y 16 maestros.
- *Processor System Reset* funciona como fuente de la señal de reset, distinguiendo entre el periférico y el módulo de interconexión.

También hay que asignar una dirección de memoria al periférico conectado al bus AXI, la cual servirá para diferenciar los periféricos y que el procesador mande al destinatario correcto la información.

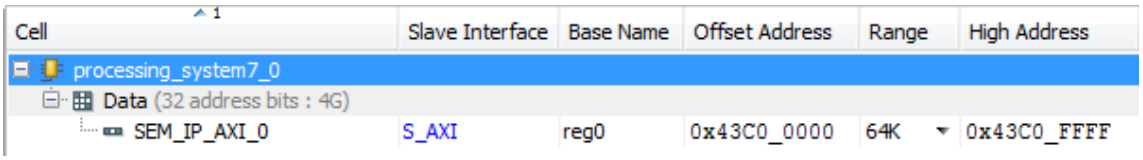

FIGURA 20: DIRECCIÓN MEMORIA SEM

Una vez creado el diseño de bloques hay que crear el *HDL Wrapper*. Este funciona como el diseño de más alto nivel escrito en VHDL, en el que un componente correspondiente al diseño de bloques es instanciado y conectado a la entidad global. En este punto se modifica el archivo VHDL para conectar las entradas y salidas del SEM a los GPIO. A su vez, se dejan abiertas las señales del *Monitor Interface* ya que el puerto UART de la placa ya está en uso.

Las conexiones de los GPIO del código VHDL están descritas en la *TABLA 1*.

GPIO nº	GPIO entrada/salida	Señal conectada
0	Entrada	Status_heartbeat
1	Entrada	Status_initialization
2	Entrada	Status_observation
3	Entrada	Status_correction
4	Entrada	Status_classification
5	Entrada	Status_injection
6	Entrada	Status_essential
7	Entrada	Status_uncorrectable
8	Salida	Inject_strobe
9	Salida	Icap_grant
10	Salida	Reset_signal_status
11	Entrada	Señal de error para el test

TABLA 1: CONEXIONES GPIO

Una vez hecho todo el diseño, hay que generar el *bitstream* previa síntesis e implementación. Realizados estos pasos, se puede exportar el proyecto incluyendo el *bitstream* al programa Xilinx SDK para el desarrollo del software.

3.4. Software de la aplicación

Una vez exportado el proyecto, se creó un proyecto de aplicación en Xilinx SDK para programar el software del sistema en C. En su programación se ha hecho uso de las funciones de librería para los periféricos incluidas en el programa. El código completo puede verse en el *APENDICE 2.1*. Aquí se explicarán las partes más importantes y críticas para su comprensión.

El programa puede dividirse en tres partes diferenciadas: un setup de inicialización, el mecanismo de inyección de fallos pseudo-aleatorio y un informe sobre la inyección. El programa es lineal y cíclico, excepto el setup que únicamente se realiza al comienzo de la ejecución.

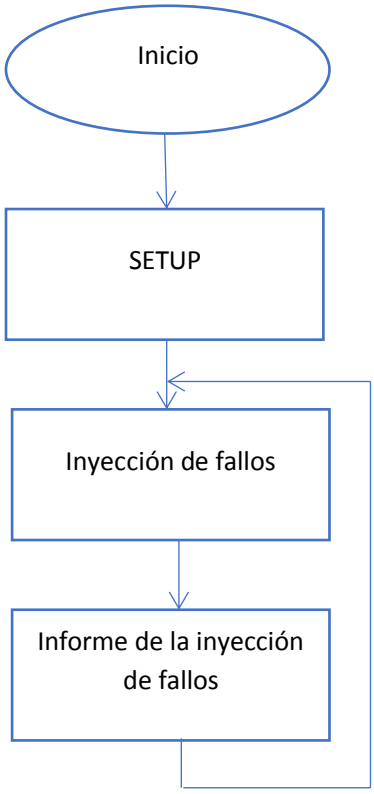

FIGURA 21: DIAGRAMA DE FLUJO DEL PROGRAMA

- SETUP

El setup inicializa todos los elementos a utilizar con las funciones de librería indicadas para ello. Además configura adecuadamente sus características, habilita interrupciones y asigna un valor inicial. Los elementos a configurar son: GPIO, UART1, Timer_0, GIC (Generic Interrupt Controller) y devcfg (Device Configuration Interface). Además, se le pide al usuario que indique un número que haga de semilla para comenzar el cálculo aleatorio de las direcciones.

En la configuración del GPIO se asignaron los puertos de entrada y salida correspondientes y se habilitaron las interrupciones por flanco de subida para las señales del interfaz de estado del SEM, excluyendo las señales status_heartbeat y status_essential que no son de interés para esta aplicación.

```
/*Funcion de inicializacion del GPIO*/
void init_GPIO()
{
    int Status;
    ConfigPtr_GPIO=XGpioPs_LookupConfig(GPIO_DEVICE_ID);
    Status = XGpioPs_CfgInitialize(&GPIO,ConfigPtr_GPIO, ConfigPtr_GPIO->BaseAddr);
        if(Status != XST_SUCCESS)
        {return XST_FAILURE;}
    /*Configuro como entrada el  Bank_2 del GPIO (señales 54 a 85 del EMIO) excepto la 62, 63 y 64*/
    XGpioPs_SetDirection(&GPIO, Bank_2, INPUT);
    XGpioPs_SetDirectionPin(&GPIO, 62, OUTPUT);//inject_strobe
    XGpioPs_SetDirectionPin(&GPIO, 63, OUTPUT);//icap_grant
    XGpioPs_SetDirectionPin(&GPIO, 64, OUTPUT);//reset_signal_status
    XGpioPs_WritePin(&GPIO, 62, 0);
    XGpioPs_WritePin(&GPIO, 63, 0);
    XGpioPs_WritePin(&GPIO, 64, 0);
    Enable_interrupt_GPIO();
}
/*Habilita las interrupciones del GPIO en el controlador de interrupciones*/
void Enable_interrupt_GPIO()
{
    int status;
    IntcConfig = XScuGic_LookupConfig(XSCUGIC_DEVICE_ID);
    status = XScuGic_CfgInitialize(&INTCInst, IntcConfig, IntcConfig->CpuBaseAddress);
    if(status != XST_SUCCESS) return XST_FAILURE;
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
            (Xil_ExceptionHandler)XScuGic_InterruptHandler, &INTCInst);
        Xil_ExceptionEnable();
    status = XScuGic_Connect(&INTCInst,INTC_GPIO_INTERRUPT_ID,
            (Xil_ExceptionHandler)GPIO_Intr_Handler,(void *)&GPIO);
    if(status != XST_SUCCESS) return XST_FAILURE;
    XScuGic_SetPriorityTriggerType(&INTCInst, INTC_GPIO_INTERRUPT_ID,PRIORITY_0, RISING_EDGE_GIC);
    XGpioPs_SetIntrType(&GPIO, Bank_2, EDGE_SENSITIVE, RISING_EDGE_GPIO, SINGLE_EDGE);
    XGpioPs_IntrEnable(&GPIO, Bank_2,0x000008BE);//status_heartbeat => GPIO(0) no activada
                                                 //status_essential => GPIO(6) no activada
    XScuGic_Enable(&INTCInst, INTC_GPIO_INTERRUPT_ID);
}
```

FIGURA 22: CONFIGURACIÓN GPIO

El devcfg hay que configurarlo adecuadamente para permitir el acceso al PL de ICAP como se describió en el *Capítulo 3.3.1.* , ya que el procesador durante la configuración de la FPGA bloquea este acceso permitiendo el acceso al PCAP. En la *FIGURA 11* puede verse como un multiplexor controla el acceso de uno u otro. Para poder controlar adecuadamente esto hay que manejar dos bits del registro de control del *Device Configuration Interface*:

Nombre	Devcfg Bit nº	Activado/Desactivado
PCAP_MODE	26	Activado
PCAP_PR	27	Desactivado

TABLA 2: CONDICIÓN DE ACCESO ICAP

Esto tiene que hacerse antes de activar la señal *icap_grant* mediante el GPIO para evitar que el controlador SEM intente acceder a la FPGA y se produzca un comportamiento inesperado. El código C que asigna estos valores está incluido en la *FIGURA 23*:

```
void init_Dcfg()
{
    int Status;

    ConfigPtr_Dcfg = XDcfg_LookupConfig(DCFG_DEVICE_ID);

    /*Inicializa el Device Config Interface*/
    Status = XDcfg_CfgInitialize(&Dcfg , ConfigPtr_Dcfg, ConfigPtr_Dcfg->BaseAddr);
    if(Status != XST_SUCCESS)
    {return XST_FAILURE;}

    /*Deshabilita PCAP_PR y activa PCAP_MODE, para permitir la inicializacion del SEM*/
    Enable_ICAP(&Dcfg);
    /*Activo la señal Icap_grant para el SEM*/
    XGpioPs_WritePin(&GPIO, 63, 1);
}

void Enable_ICAP(XDcfg *InstancePtr)
{
    u32 CtrlReg;

    Xil_AssertVoid(InstancePtr != NULL);
    Xil_AssertVoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);

    CtrlReg = XDcfg_ReadReg(InstancePtr->Config.BaseAddr,XDCFG_CTRL_OFFSET);

    /*Escribe en el Registro de Control los valores para activar PCAP_MODE y desactivar PCAP_PR*/
    XDcfg_WriteReg(InstancePtr->Config.BaseAddr, XDCFG_CTRL_OFFSET,(CtrlReg | PCAP_MODE_MASK));
    XDcfg_WriteReg(InstancePtr->Config.BaseAddr, XDCFG_CTRL_OFFSET,(CtrlReg & ( ~PCAP_PR_MASK)));

}
```

FIGURA 23: CONFIGURACIÓN PARA ACCESO ICAP

- INYECCIÓN DE FALLOS

Una vez inicializado el controlador SEM, el proceso comienza en el estado de observación, en el que se activa el timer para esperar si el SEM detecta algún error. En la primera ejecución no tiene efecto alguno ya que no debería detectar errores. Al desbordarse el timer se activa un flag (i=1) que provoca el paso a estado idle.

Para pasar al estado idle, se manda un comando por el interfaz de inyección de error y se activa la señal *inject_strobe*. El comando formado por 40 bits tiene el siguiente formato:

1110	**XXXX**	**XXXX**	**XXXX**	**XXXX**	**XXXX**	**XXXX**	**XXXX**	**XXXX**	**XXXX**

TABLA 3: COMANDO PASO ESTADO IDLE

Una vez en el estado idle se procede a calcular la dirección de error y mandársela al SEM. Para el cálculo pseudo-aleatorio de la dirección de error se utiliza la función rand(). La dirección ha de mandarse en dos fragmentos, según se explicó en el *Apartado 3.3.1.*, ya que al ser una señal de 40 bits y el bus AXI de 32 bits, no se puede mandar de una sola vez.

El código para generar las direcciones de error y mandarlas por el bus AXI se muestra en la *FIGURA 24*:

```
//Idle State
case 2:
    Inj_sup=Calculo_inject_address();   //Calculo las nuevas direcciones
    Inj_inf=Calculo_inject_address();
    Write_inject_address(Inj_sup, Inj_inf);     //Escribo las direcciones en el SEM
    cuenta_errores++;
    error_corregido=0;
    i=4;
    XScuTimer_LoadTimer(&TIMER, Val_Timer2);
    XScuTimer_Start(&TIMER);
    break;
```

FIGURA 24: CÓDIGO INYECCIÓN DE ERROR

La función Write_inject_address contiene el proceso de escritura de la dirección de error. Mediante máscaras se escribe '0' o '1' en el bit 32 del valor que se va a mandar por el interfaz AXI al SEM y así indicar si el valor corresponde a la parte superior o inferior de la dirección de error.

```
void Write_inject_address(u32 superior, u32 inferior)
{
    u32 superior_bis;
    superior_bis=(~INJECT_MASK & superior);
    Write_error_address(BIT_32_MASK |superior_bis);
    Delay(200);
    Write_error_address(~BIT_32_MASK & inferior);
    Delay(20);
    activar_inject_strobe();
    Delay(20);
    desactivar_inject_strobe();
}
```

FIGURA 25: FUNCIÓN WRITE_INJECT_ADDRESS

Mediante la función rand() se calcula de forma pseudo-aleatoria el valor de la dirección de inyección y se devuelve su valor.

```
u32 Calculo_inject_address()
{
    u32 inject_address;

    inject_address=rand();
    return inject_address;
}
```

FIGURA 26: FUNCIÓN CALCULO_INJECT_ADDRESS

Para escribir o leer la información mandada por el bus AXI a la señal *inject_address* se han utilizado las funciones Xil_Out32 y Xil_In32 del driver "xil_io". Estas funciones son utilizadas para mandar o pedir un dato de 32 bits a un periférico indicando su dirección de memoria.

Para mandar el comando con la dirección de error, se utiliza el método "Linear Frame Address", que consiste en un comando con el siguiente formato:

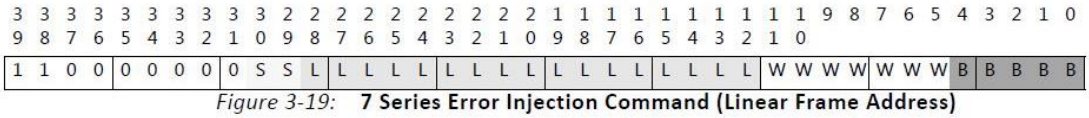

Figure 3-19: **7 Series Error Injection Command (Linear Frame Address)**

Where:

- SS = Hardware SLR number for SSI (2-bit) and set to 00 for non-SSI
- LLLLLLLLLLLLLLLLL = linear frame address (17-bit)
- WWWWWWW = word address (7-bit)
- BBBBB = bit address (5-bit)

FIGURA 27: COMANDO INYECCIÓN DE ERROR

Tras inyectar el error el controlador SEM vuelve automáticamente al estado idle. En este punto se manda de nuevo un comando para pasar al estado de observación. El comando de 40 bits tiene la siguiente forma:

1010	**XXXX**	**XXXX**	**XXXX**	**XXXX**	**XXXX**	**XXXX**	**XXXX**	**XXXX**	**XXXX**

TABLA 4: COMANDO PASO ESTADO OBSERVACIÓN

De nuevo en el estado de observación, se activa el timer y se espera para volver a realizar las operaciones. En este periodo de tiempo el controlador SEM está observando la FPGA en busca de cambios que denoten un error. Si encuentra alguno el SEM pasa al estado de corrección y salta la interrupción del GPIO correspondiente, parando la ejecución del timer y permitiendo al SEM intentar corregir el error.

Si el error se corrige, se volverá a activar el timer y se reinicializará el proceso hasta que se encuentre un error incorregible o se pare la ejecución del programa. En caso de no poder corregirse el error, saltaría la interrupción del GPIO correspondiente a la señal *status_uncorrectable* y se finalizaría la campaña de inyección de errores, dando paso a la emisión del informe.

Para controlar más eficazmente el paso de un estado a otro, se incluye una espera de tiempo marcada por el timer. Esto se hace necesario ya que la velocidad de reloj del microprocesador es bastante mayor que la del SEM y este necesita un cierto número de ciclos para completar la tarea indicada para cada estado. Por esta razón se controla mediante el desbordamiento del timer los tiempos de ejecución del programa.

- INFORME DE INYECCIÓN DE FALLOS

Tras terminar la campaña de inyección de fallos y cuando encuentra un error y lo corrige, se procede a mandar a través de la UART1 un informe. Este indica el número de errores corregidos, incorregibles y totales además de la dirección de error aleatoria que fue generada en los casos que el error fuera corregido o no, pero que tuviera algún efecto sobre la FPGA.

La comunicación máquina-usuario se realiza con el puerto USB-UART de la placa Zedboard conectado al PC y en éste trabajando un emulador de terminal. Por su facilidad de uso y por ser un programa open source se trabajó con Tera Term con una comunicación serie a 115200 baudios.

El código generador del informe es el siguiente:

```
if(report!=0)
{
    report=0;
    if(impresion_errores==0)
    {
        if(error_corregido==1)
        {
        printf("Error %d: \n\rDireccion superior: %d\n\rDireccion inferior: %d\n\r"
            ,cuenta_errores,Inj_sup,Inj_inf);
        printf("CORREGIDO\n\r");
        if(error_circuito_test==1)
        {
            error_circuito_test=0;
            printf("Este error pertenece al circuito en test\n\r");
            error_test++;
        }
        total_corregidos++;
        }
        if(error_corregido==2)
        {
            printf("Error %d: \n\rDireccion superior: %d\n\rDireccion inferior: %d\n\r"
                ,cuenta_errores,Inj_sup,Inj_inf);
            printf("INCORREGIBLE\n\r");
            if(error_circuito_test==1)
            {
                error_circuito_test=0;
                printf("Este error pertenece al circuito en test\n\r");
                error_test++;
            }
            total_incorregibles++;
            impresion_errores=1;
            printf("\n\rTotal errores corregidos:%d\n\r",total_corregidos);
            printf("Total errores incorregibles:%d\n\r", total_incorregibles);
            printf("Total errores en el circuito de test:%d\n\r", error_test);
            printf("Total errores inyectados:%d\n\r",cuenta_errores);
            printf("\n\rReinicie la FPGA para inyectar mas fallos\n\r");
        }
```

FIGURA 28: CÓDIGO INFORME DE ERRORES

4.Validación experimental

Durante el desarrollo del proyecto se han ido realizando pruebas con distintos circuitos para comprobar el funcionamiento y mejorar la aplicación, siempre con el objetivo de depurar errores y validar el funcionamiento. Una vez conseguido un diseño satisfactorio se ha realizado una campaña de inyección de fallos en un circuito de prueba replicado varias veces, en donde las salidas de cada una de las réplicas se comparan con un valor correcto esperado. El resultado de esta comparación es una señal de error, que se activa cuando al menos una de las réplicas produce un valor incorrecto. Se utilizan varias réplicas para ocupar un mayor espacio en la FPGA y poder observar un mayor número de errores.

El circuito sobre el que probar la aplicación es el circuito B13 del conjunto de circuitos de prueba ("benchmarks") denominado ITC'99 (International Test Conference, 1999) [11]. Este conjunto de circuitos de prueba se utilizan habitualmente en numerosos estudios en el ámbito del test y el diseño de circuitos integrados. El circuito B13 es un interfaz para un sensor. Este benchmark ha sido facilitado por el tutor del trabajo, incluyendo un generador de entradas y un comparador de salidas. El circuito se ejercita durante 7369 ciclos de reloj y al finalizar comprueba si ha habido alguna diferencia, activando la señal de error en caso afirmativo.

El número de réplicas se puede variar mediante una constante, que para conseguir un porcentaje de ocupación mayor de la FPGA y de esta manera tener una mayor probabilidad de obtener un error en el circuito, se ha fijado en 500.

De esta manera se ha obtenido un porcentaje de ocupación entorno a un 64% de LUTs. Siendo este el parámetro critico a la hora de poder aumentar el número de réplicas. No se ha querido aumentar el porcentaje de ocupación más, para no generar posibles inconvenientes a la hora de rutar el diseño.

Name	Slice LUTs* (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Block RAM Tile (140)	Bonded IOPADs (130)	BUFGCTRL (32)	FRAME_ECCE2 (1)	ICAPE2 (2)
TFG_wrapper	64.29 %	22.51 %	0.06 %	0.04 %	13.92 %	100.00 %	12.50 %	100.00 %	50.00 %
b_13 (wrap_b13)	61.83 %	21.26 %	0.00 %	0.00 %	10.71 %	0.00 %	0.00 %	0.00 %	0.00 %
TFG_i (TFG)	2.45 %	1.25 %	0.06 %	0.04 %	3.21 %	0.00 %	12.50 %	100.00 %	50.00 %

FIGURA 29: OCUPACIÓN DE LA FPGA

Al realizar las pruebas se pueden encontrar distintos resultados ante un error inyectado en la memoria de configuración. Estos pueden dividirse en:

- Errores corregidos: el error inyectado es detectado y corregido por el controlador SEM. El programa emite un informe y sigue su curso inyectando más errores.
- Errores incorregibles: el error inyectado es detectado por el controlador SEM pero no puede corregirlo. El SEM activa la señal *status_uncorrectable*. El programa no sigue inyectando errores. Hay que reconfigurar la FPGA.
- Errores sin efecto: el controlador SEM no detecta ningún error en la memoria de configuración. El flujo del programa sigue su curso inyectando más errores.

Como complemento a estas categorías de error, se tiene el resultado de la comparación de la salida de las réplicas. Esta señal conectada a un GPIO indica cuando se ha encontrado un error en el circuito de prueba. Este se ha denominado error en el circuito en test.

Para realizar las pruebas lo primero que hay que hacer es cargar el diseño en la FPGA, esto se realiza mediante el programa Xilinx SDK. Al cargar el diseño el programa sigue un orden claro: primero inicializa el sistema completo, después programa la FPGA o PL y posteriormente inicia y carga la configuración para el procesador o PS. Estos pasos son configurables y se pueden suprimir o modificar algunos de ellos, además de otras opciones.

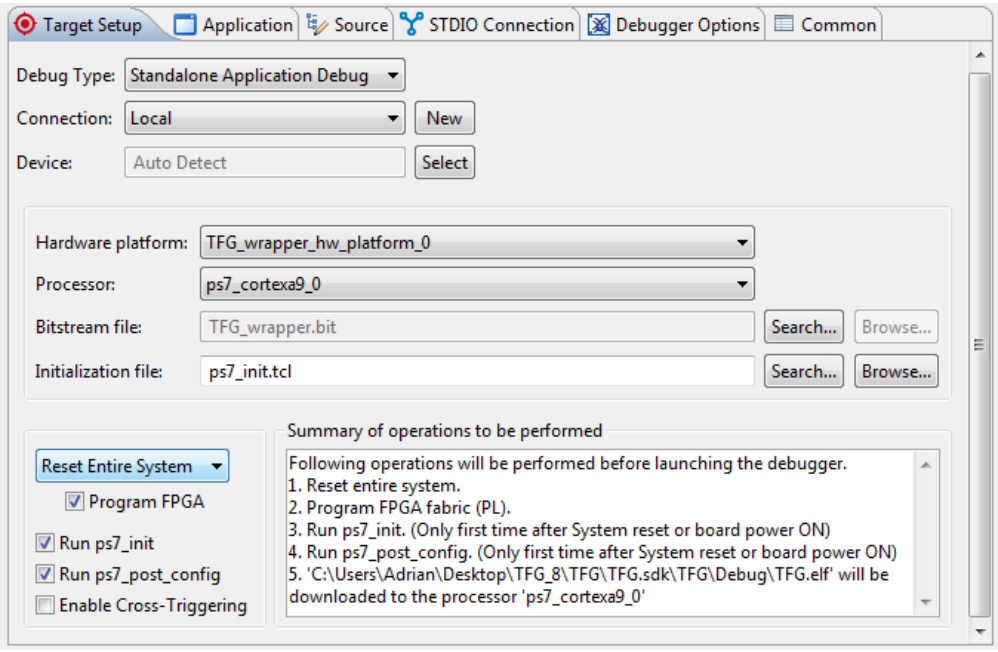

FIGURA 30: CONFIGURACIÓN DE DEBUG EN XILINX SDK

También hay que configurar un terminal serie como Tera Term para la comunicación máquina-usuario. Para ello se conecta el puerto USB-UART de la placa Zedboard al PC y eligiendo el puerto adecuado, se configura con los mismos parámetros que la UART en Vivado. La *FIGURA 31* indica cómo queda la configuración.

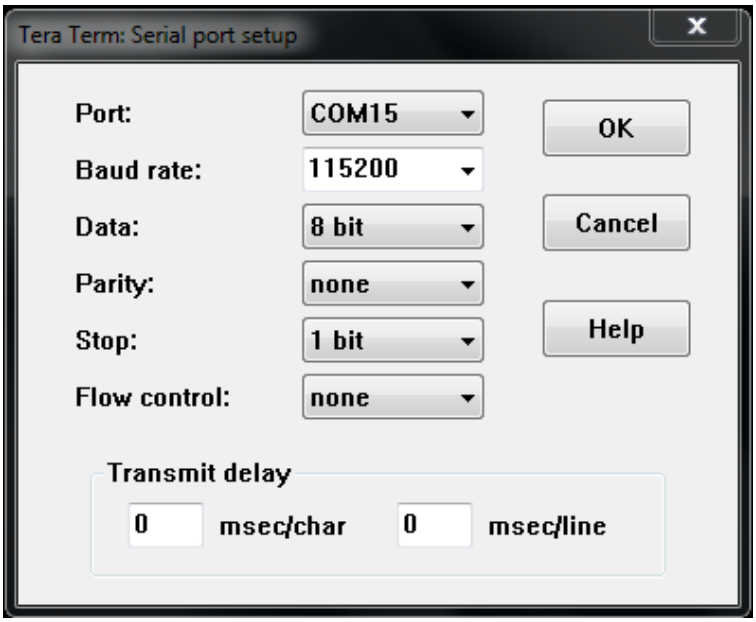

FIGURA 31: CONFIGURACIÓN PUERTO SERIE EN TERA TERM

Una vez preparado todo, se puede ejecutar la aplicación. Esta inyectará los fallos automáticamente hasta que encuentre un error incorregible, con lo que finalizará la ejecución. Para inyectar fallos en distintas direcciones de memoria y no repetir siempre las mismas, se indicará un número distinto para la semilla de la función srand() cada vez que se ejecute.

Este es un ejemplo del informe que genera la aplicación al inyectar errores habiendo elegido un valor de semilla igual a 3:

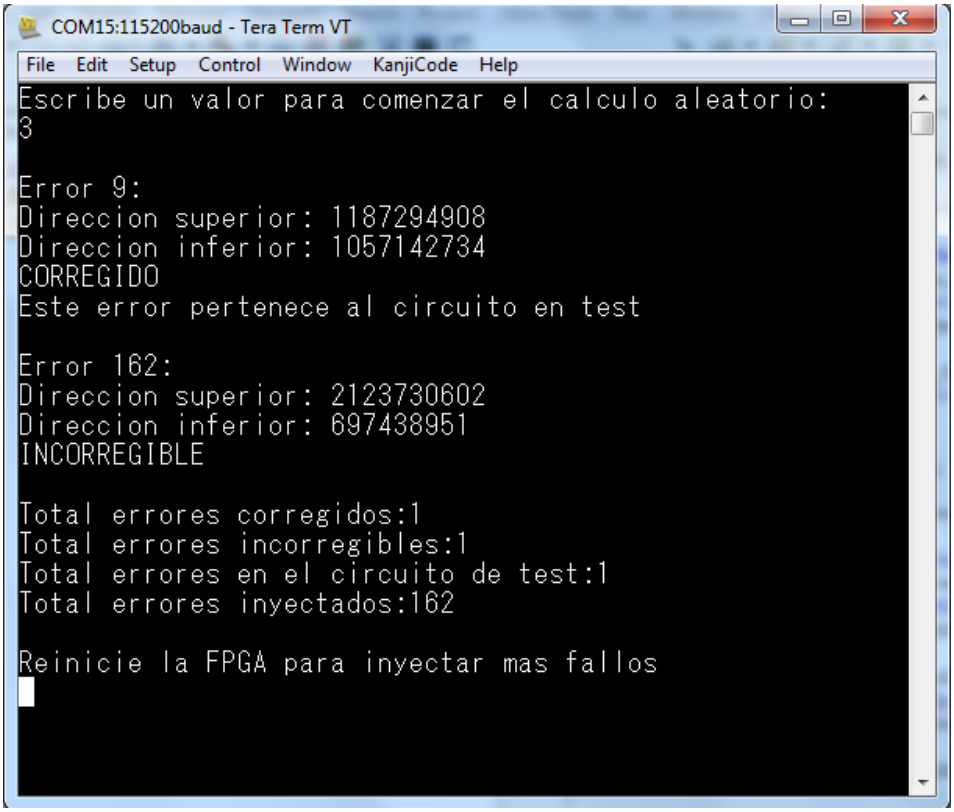

FIGURA 32: INFORME DE ERRORES

Cabe destacar que para conocer la dirección de inyección, hay que coger los 20 bits menos significativos de las direcciones indicadas en el informe y juntarlas, ya que la dirección se manda en dos fracciones dada la imposibilidad de mandar los 40 bits por el bus AXI de 32 bits.

La *TABLA 5* indica según un valor de semilla aleatorio, el número de errores que se han inyectado, cuantos se han corregido, cuantos no se han podido corregir y cuantos pertenecían al circuito en test.

Valor de semilla	Errores totales inyectados	Errores corregidos	Errores incorregibles	Errores circuito en test
3	162	1	1	1
6	680	2	1	0
9	1181	8	1	3
15	391	4	1	3
21	712	3	1	1
24	659	5	1	2
32	98	0	1	1
37	621	4	1	2
40	255	2	1	1
46	847	6	1	3

TABLA 5: RECOPILACIÓN ERRORES

Como puede verse, solo un pequeño número del total de errores inyectados provoca errores que tengan que ser corregidos por el controlador SEM. Esto es debido a la existencia de bits de la memoria de configuración que no se usan. Aun así puede comprobarse que el SEM inyecta y es capaz de corregir errores. También se aprecia cómo se activa la señal de error del circuito de prueba, al haber encontrado una diferencia entre las salidas de cada réplica.

5.Conclusiones

El objetivo de este trabajo consistía en desarrollar una aplicación que permitiera inyectar fallos y corregirlos utilizando el controlador SEM de Xilinx. Para ello se utilizaría una FPGA tipo Zynq, en concreto la placa Zedboard, con un microprocesador embebido para controlar el sistema.

Con este propósito, lo primero fue familiarizarse con la tecnología Zynq, aprendiendo sus partes y su funcionamiento. A su vez, se realizó el mismo estudio con el controlador SEM. Además de aprender a utilizar las herramientas de diseño de Xilinx, Vivado y Xilinx SDK, realizando aplicaciones sencillas y tutoriales.

Una vez adquiridos los primeros conocimientos, se comenzó a desarrollar la aplicación. En este proceso se encontraron multitud de problemas e inconvenientes que fueron solventándose hasta obtener el resultado descrito en esta memoria.

En este trabajo se desarrollan partes hardware y software, teniendo que poner en común ambos diseños lo que genera una dificultad extra en su desarrollo, debido a que hasta ahora siempre había trabajado en una u otra de las partes únicamente. Al desarrollar ambas partes, los diseños hardware hay que enfocarlos al posterior control por parte del microprocesador, implementando los interfaces a utilizar.

Para el desarrollo del trabajo se han puesto en uso gran cantidad de conocimientos obtenidos en distintas asignaturas del grado, como conocimientos sobre electrónica digital, FPGAs, microprocesadores, C o VHDL. Pero a su vez se ha tenido que aprender o ampliar conocimientos nuevos, como el trabajo con IP.

Aunque este trabajo cumple con los objetivos iniciales, podrían realizarse ciertas mejoras de interés. Entre las mejoras aplicables se encuentra el desarrollo de un sistema que permita una auto-reconfiguración de la FPGA una vez que el SEM encuentra un error incorregible. Este sistema conllevaría tener almacenada la configuración en una memoria externa, que podría ser por ejemplo una tarjeta SD conectada al lector de la placa Zedboard. Esto aportaría mayor rapidez al evitar reconfigurar manualmente la FPGA.

6.Presupuesto

Código	Uds. Medida	Descripción	Cantidad	Precio/Ud. [€]	Importe [€]
6.1. COSTES DE MATERIAL					
6.1.1.	Uds.	ZedBoard Evaluation Kit FPGA tipo Zynq de Xilinx, desarrollada y comercializada por AVNET.	1	348,41	348,41
6.1.2.	Uds.	Ordenador Personal Procesador Intel Core 2 Duo, Memoria RAM 4GB, Disco Duro 500GB, Windows 7, conexión Wifi.	1	249,00	249,00
6.1.3.	Uds.	Vivado Design Suite Software de diseño, compatible con tecnología Zynq, acepta lenguaje VHDL y C, licencia gratuita.	1	0,00	0,00
6.1.4.	Uds.	Tera Term Terminal emulator, comunicación serie, software open source.	1	0,00	0,00
				SUBTOTAL COSTES DE MATERIAL	597,41
6.2. COSTES DE PERSONAL					
6.2.1.	Meses	Ingeniero Ingeniero electrónico industrial, conocimientos de programación C, programación VHDL, electrónica digital, microprocesadores y FPGAs.	4	1500,00	6000,00
				SUBTOTAL COSTES DE PERSONAL	6000,00
				TOTAL PROYECTO	6597,41

TABLA 6: PRESUPUESTO DEL PROYECTO

APÉNDICES

1.Código VHDL del proyecto

1.1. sem_0_sem_example.vhd

Modificación del archivo original del ejemplo para el IP del controlador SEM. No se muestran aquí los archivos no modificados a los que hace referencia el código.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

library unisim;
use unisim.vcomponents.all;

------------------------------------------------------------------------------
-- Entity
------------------------------------------------------------------------------

entity sem_0_sem_example is
port (
  clk                           : in    std_logic;
  status_heartbeat              : out   std_logic;
  status_initialization         : out   std_logic;
  status_observation            : out   std_logic;
  status_correction             : out   std_logic;
  status_classification         : out   std_logic;
  status_injection              : out   std_logic;
  status_essential              : out   std_logic;
  status_uncorrectable          : out   std_logic;
  reset_signal_status           : in    std_logic;
  inject_strobe                 : in    std_logic;
  inject_address                : in    std_logic_vector(39 downto 0);
  icap_grant                    : in    std_logic;
  monitor_tx                    : out   std_logic;
  monitor_rx                    : in    std_logic
  );
end entity sem_0_sem_example;
```

```
------------------------------------------------------------------------------
-- Architecture
------------------------------------------------------------------------------

architecture xilinx of sem_0_sem_example is

  ------------------------------------------------------------------------------
  -- Define local constants.
  ------------------------------------------------------------------------------

  constant TCQ : time := 1 ps;

  ------------------------------------------------------------------------------
  -- Declare non-library components.
  ------------------------------------------------------------------------------

  component sem_0
  port (
    status_heartbeat            : out   std_logic;
    status_initialization       : out   std_logic;
    status_observation          : out   std_logic;
    status_correction           : out   std_logic;
    status_classification       : out   std_logic;
    status_injection            : out   std_logic;
    status_essential            : out   std_logic;
    status_uncorrectable        : out   std_logic;
    monitor_txdata              : out   std_logic_vector (7 downto 0);
    monitor_txwrite             : out   std_logic;
    monitor_txfull              : in    std_logic;
    monitor_rxdata              : in    std_logic_vector (7 downto 0);
    monitor_rxread              : out   std_logic;
    monitor_rxempty             : in    std_logic;
    inject_strobe               : in    std_logic;
    inject_address              : in    std_logic_vector (39 downto 0);
    fecc_crcerr                 : in    std_logic;
    fecc_eccerr                 : in    std_logic;
    fecc_eccerrsingle           : in    std_logic;
    fecc_syndromevalid          : in    std_logic;
    fecc_syndrome               : in    std_logic_vector (12 downto 0);
    fecc_far                    : in    std_logic_vector (25 downto 0);
    fecc_synbit                 : in    std_logic_vector (4 downto 0);
    fecc_synword                : in    std_logic_vector (6 downto 0);
    icap_o                      : in    std_logic_vector (31 downto 0);
    icap_i                      : out   std_logic_vector (31 downto 0);
    icap_csib                   : out   std_logic;
    icap_rdwrb                  : out   std_logic;
    icap_clk                    : in    std_logic;
    icap_request                : out   std_logic;
    icap_grant                  : in    std_logic
    );
  end component;
```

```
component sem_0_sem_cfg
port (
  fecc_crcerr                : out   std_logic;
  fecc_eccerr                : out   std_logic;
  fecc_eccerrsingle          : out   std_logic;
  fecc_syndromevalid         : out   std_logic;
  fecc_syndrome              : out   std_logic_vector (12 downto 0);
  fecc_far                   : out   std_logic_vector (25 downto 0);
  fecc_synbit                : out   std_logic_vector (4 downto 0);
  fecc_synword               : out   std_logic_vector (6 downto 0);
  icap_o                     : out   std_logic_vector (31 downto 0);
  icap_i                     : in    std_logic_vector (31 downto 0);
  icap_clk                   : in    std_logic;
  icap_csib                  : in    std_logic;
  icap_rdwrb                 : in    std_logic
  );
end component;

component sem_0_sem_mon
port (
  icap_clk                   : in    std_logic;
  monitor_tx                 : out   std_logic;
  monitor_rx                 : in    std_logic;
  monitor_txdata             : in    std_logic_vector (7 downto 0);
  monitor_txwrite            : in    std_logic;
  monitor_txfull             : out   std_logic;
  monitor_rxdata             : out   std_logic_vector (7 downto 0);
  monitor_rxread             : in    std_logic;
  monitor_rxempty            : out   std_logic
  );
end component;

-- ------------------------------------------------------------------------
-- Declare signals.
-- ------------------------------------------------------------------------

signal clk_ibufg : std_logic;

signal status_heartbeat_internal : std_logic;
signal status_initialization_internal : std_logic;
signal status_observation_internal : std_logic;
signal status_correction_internal : std_logic;
signal status_classification_internal : std_logic;
signal status_injection_internal : std_logic;
signal status_essential_internal : std_logic;
signal status_uncorrectable_internal : std_logic;
```

```
signal monitor_txdata         : std_logic_vector(7 downto 0);
signal monitor_txwrite        : std_logic;
signal monitor_txfull         : std_logic;
signal monitor_rxdata         : std_logic_vector (7 downto 0);
signal monitor_rxread         : std_logic;
signal monitor_rxempty        : std_logic;
signal fecc_crcerr            : std_logic;
signal fecc_eccerr            : std_logic;
signal fecc_eccerrsingle      : std_logic;
signal fecc_syndromevalid     : std_logic;
signal fecc_syndrome          : std_logic_vector (12 downto 0);
signal fecc_far               : std_logic_vector (25 downto 0);
signal fecc_synbit            : std_logic_vector (4 downto 0);
signal fecc_synword           : std_logic_vector (6 downto 0);
signal icap_o                 : std_logic_vector (31 downto 0);
signal icap_i                 : std_logic_vector (31 downto 0);
signal icap_csib              : std_logic;
signal icap_rdwrb             : std_logic;
signal icap_unused            : std_logic;
signal icap_clk               : std_logic;

signal icap_grant_parcial     : std_logic;
------------------------------------------------------------------------------
--
------------------------------------------------------------------------------

begin

------------------------------------------------------------------------------
-- This design (the example, including the controller itself) is fully
-- synchronous; the global clock buffer is instantiated here to drive
-- the icap_clk signal.
------------------------------------------------------------------------------

example_ibuf : IBUF
port map (
  I => clk,
  O => clk_ibufg
  );

example_bufg : BUFGCE
port map (
  I => clk_ibufg,
  O => icap_clk,
  CE => '1'
  );
```

```
-------------------------------------------------------------------------------
-- The controller sub-entity is the kernel of the soft error mitigation
-- solution.  The port list is dynamic based on the IP core options.
-------------------------------------------------------------------------------

example_controller : sem_0
port map (
  status_heartbeat => status_heartbeat_internal,
  status_initialization => status_initialization_internal,
  status_observation => status_observation_internal,
  status_correction => status_correction_internal,
  status_classification => status_classification_internal,
  status_injection => status_injection_internal,
  status_essential => status_essential_internal,
  status_uncorrectable => status_uncorrectable_internal,
  monitor_txdata => monitor_txdata,
  monitor_txwrite => monitor_txwrite,
  monitor_txfull => monitor_txfull,
  monitor_rxdata => monitor_rxdata,
  monitor_rxread => monitor_rxread,
  monitor_rxempty => monitor_rxempty,
  inject_strobe => inject_strobe,
  inject_address => inject_address,
  fecc_crcerr => fecc_crcerr,
  fecc_eccerr => fecc_eccerr,
  fecc_eccerrsingle => fecc_eccerrsingle,
  fecc_syndromevalid => fecc_syndromevalid,
  fecc_syndrome => fecc_syndrome,
  fecc_far => fecc_far,
  fecc_synbit => fecc_synbit,
  fecc_synword => fecc_synword,
  icap_o => icap_o,
  icap_i => icap_i,
  icap_csib => icap_csib,
  icap_rdwrb => icap_rdwrb,
  icap_clk => icap_clk,
  icap_request => icap_unused,
  icap_grant => icap_grant_parcial
  );

PROCESS(clk_ibufg)
BEGIN
IF clk_ibufg'EVENT AND clk_ibufg='1' THEN
    icap_grant_parcial <= icap_grant;
END IF;
END PROCESS;
```

```
--------------------------------------------------------------------------
--Mantengo las señales activas el suficiente tiempo para que el programa
--software pueda realizer la interrupcion y de esta manera conseguir un
--correcto funcionamiento del SEM
--------------------------------------------------------------------------
PROCESS (clk_ibufg)
BEGIN
IF clk_ibufg'EVENT AND clk_ibufg='1' THEN
    IF status_correction_internal='1' THEN
        status_correction<=status_correction_internal;
    ELSE
        IF reset_signal_status='1' THEN
            status_correction<='0';
        END IF;
    END IF;
END IF;
END PROCESS;

PROCESS(clk_ibufg)
BEGIN
IF clk_ibufg'EVENT AND clk_ibufg='1' THEN
    IF status_classification_internal='1' THEN
        status_classification<=status_classification_internal;
    ELSE
        IF reset_signal_status='1' THEN
            status_classification<='0';
        END IF;
    END IF;
END IF;
END PROCESS;

PROCESS(clk_ibufg)
BEGIN
IF clk_ibufg'EVENT AND clk_ibufg='1' THEN
    IF status_essential_internal='1' THEN
        status_essential<=status_essential_internal;
    ELSE
        IF reset_signal_status='1' THEN
            status_essential<='0';
        END IF;
    END IF;
END IF;
END PROCESS;
```

```
  PROCESS(clk_ibufg)
  BEGIN
  IF clk_ibufg'EVENT AND clk_ibufg='1' THEN
      IF status_uncorrectable_internal='1' THEN
          status_uncorrectable<=status_uncorrectable_internal;
      ELSE
          IF reset_signal_status='1' THEN
              status_uncorrectable<='0';
          END IF;
      END IF;
  END IF;
  END PROCESS;

status_heartbeat <= status_heartbeat_internal;
status_initialization <= status_initialization_internal;
status_observation <= status_observation_internal;
status_injection <= status_injection_internal;


------------------------------------------------------------------------------
-- The cfg sub-entity contains the device specific primitives to access
-- the internal configuration port and the frame crc/ecc status signals.
------------------------------------------------------------------------------

example_cfg : sem_0_sem_cfg
port map (
  fecc_crcerr => fecc_crcerr,
  fecc_eccerr => fecc_eccerr,
  fecc_eccerrsingle => fecc_eccerrsingle,
  fecc_syndromevalid => fecc_syndromevalid,
  fecc_syndrome => fecc_syndrome,
  fecc_far => fecc_far,
  fecc_synbit => fecc_synbit,
  fecc_synword => fecc_synword,
  icap_o => icap_o,
  icap_i => icap_i,
  icap_csib => icap_csib,
  icap_rdwrb => icap_rdwrb,
  icap_clk => icap_clk
  );
```

```
  ------------------------------------------------------------------------------
  -- The mon sub-entity contains a UART for communication purposes.
  ------------------------------------------------------------------------------

  example_mon : sem_0_sem_mon
  port map (
    icap_clk => icap_clk,
    monitor_tx => monitor_tx,
    monitor_rx => monitor_rx,
    monitor_txdata => monitor_txdata,
    monitor_txwrite => monitor_txwrite,
    monitor_txfull => monitor_txfull,
    monitor_rxdata => monitor_rxdata,
    monitor_rxread => monitor_rxread,
    monitor_rxempty => monitor_rxempty
    );


end architecture xilinx;
```

1.2. SEM_IP_AXI.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity SEM_IP_AXI is
    Port (
            -- Clock and Reset
            S_AXI_ACLK : in STD_LOGIC;
            S_AXI_ARESETN : in STD_LOGIC;
            -- Write Address Channel
            S_AXI_AWADDR : in STD_LOGIC_VECTOR (31 downto 0);
            S_AXI_AWVALID : in STD_LOGIC;
            S_AXI_AWREADY : out STD_LOGIC;
            -- Write Data Channel
            S_AXI_WDATA : in STD_LOGIC_VECTOR (31 downto 0);
            S_AXI_WSTRB : in STD_LOGIC_VECTOR (3 downto 0);
            S_AXI_WVALID : in STD_LOGIC;
            S_AXI_WREADY : out STD_LOGIC;
            -- Read Address Channel
            S_AXI_ARADDR : in STD_LOGIC_VECTOR (31 downto 0);
            S_AXI_ARVALID : in STD_LOGIC;
            S_AXI_ARREADY : out STD_LOGIC;
            -- Read Data Channel
            S_AXI_RDATA : out STD_LOGIC_VECTOR (31 downto 0);
            S_AXI_RRESP : out STD_LOGIC_VECTOR (1 downto 0);
            S_AXI_RVALID : out STD_LOGIC;
            S_AXI_RREADY : in STD_LOGIC;
            -- Write Respnse Channel
            S_AXI_BRESP : out STD_LOGIC_VECTOR (1 downto 0);
            S_AXI_BVALID : out STD_LOGIC;
            S_AXI_BREADY : in STD_LOGIC;

            inject_address : out STD_LOGIC_VECTOR (39 downto 0)
              );
end SEM_IP_AXI;



architecture Behavioral of SEM_IP_AXI is

signal address_valid_write : STD_LOGIC;
signal address_valid_read : STD_LOGIC;
signal write_registers : STD_LOGIC;
signal read_registers : STD_LOGIC;
signal dato_leido : STD_LOGIC_VECTOR(31 downto 0);
signal comb_S_AXI_AWVALID_S_AXI_ARVALID : STD_LOGIC_VECTOR(1 downto 0);
```

```
signal direccion_escritura : INTEGER;
signal direccion_lectura : INTEGER;

TYPE estado IS (reset,idle,write_transition,read_transition,completo);
SIGNAL Estado_actual : estado;
SIGNAL Proximo_estado : estado;

begin

comb_S_AXI_AWVALID_S_AXI_ARVALID <= S_AXI_AWVALID & S_AXI_ARVALID;

--Registro para escribir el dato
process(S_AXI_ACLK)
begin
IF S_AXI_ACLK='1' AND S_AXI_ACLK'EVENT THEN
    IF S_AXI_ARESETN='0' THEN   --activo por nivel bajo
        inject_address(39 downto 0)<=X"0000000000";
    ELSE
            IF dato_leido(31)='0' THEN
                inject_address(19 downto 0)<=dato_leido(19 downto 0);
            ELSE
                inject_address(39 downto 20)<=dato_leido(19 downto 0);
            END IF;
   END IF;
END IF;
end process;

--leer un dato
process(address_valid_read, read_registers, dato_leido)
begin
S_AXI_RDATA<=(others=>'-');
IF address_valid_read='1' AND read_registers ='1' THEN
    S_AXI_RDATA<=dato_leido;
END IF;
end process;


direccion_escritura<=CONV_INTEGER(unsigned(S_AXI_AWADDR));
--Comprobacion direccion de escritura
process(direccion_escritura, write_registers)
begin
if write_registers='1' then
    case direccion_escritura is
        when 1 to 99=>  address_valid_write<='1';
        when others=> address_valid_write<='0';
    end case;
end if;
end process;
```

```
direccion_lectura<=CONV_INTEGER(unsigned(S_AXI_ARADDR));
--Comprobacion direccion de lectura
process(direccion_lectura)
begin
    case direccion_lectura is
        when 1 to 99=>  address_valid_read<='1';
        when others=> address_valid_read<='0';
    end case;
end process;

--Paso de un estado a otro e inicialización
PROCESS(S_AXI_ACLK)
BEGIN
IF S_AXI_ACLK='1' AND S_AXI_ACLK'EVENT THEN
    IF S_AXI_ARESETN='0' THEN   --activo por nivel bajo
        Estado_actual<=reset;
    ELSE
        Estado_actual<=Proximo_estado;
    END IF;
END IF;
end process;

--Proceso de escritura y lectura, estados
process(Estado_actual,comb_S_AXI_AWVALID_S_AXI_ARVALID,
S_AXI_RREADY,S_AXI_BREADY,S_AXI_ARVALID,S_AXI_AWVALID,S_AXI_WVALID,S_AXI_WDAT
A)
begin
    S_AXI_AWREADY<='0';
    S_AXI_ARREADY<='0';
    S_AXI_RVALID<='0';
    S_AXI_RRESP<="--";
    S_AXI_WREADY<='0';
    S_AXI_BRESP<="--";
    S_AXI_BVALID<='0';
    write_registers<='0';
    read_registers<='0';

    case Estado_actual is
        when reset =>
            Proximo_estado<=idle;


        when idle =>
            Proximo_estado<=idle;
            case comb_S_AXI_AWVALID_S_AXI_ARVALID is
                when "10"=>Proximo_estado<=write_transition;
                when "01"=>Proximo_estado<=read_transition;
                when others=>NULL;
            end case;
```

```
        when read_transition =>
            Proximo_estado<=read_transition;
            S_AXI_ARREADY<=S_AXI_ARVALID;
            S_AXI_RVALID<='1';
            S_AXI_RRESP<="00";
            read_registers<='1';
            if S_AXI_RREADY='1' then
                Proximo_estado<= completo;
            end if;

        when write_transition =>
            Proximo_estado<=write_transition;
            S_AXI_AWREADY<=S_AXI_AWVALID;
            S_AXI_WREADY<=S_AXI_WVALID;
            S_AXI_BRESP<="00";
            S_AXI_BVALID<='1';
            write_registers<='1';
            dato_leido<=S_AXI_WDATA;
            IF S_AXI_BREADY='1' THEN
                Proximo_estado<=completo;
            END IF;

        when completo=>
            case comb_S_AXI_AWVALID_S_AXI_ARVALID is
                when "00" => Proximo_estado<=idle;
                when others =>Proximo_estado<=completo;
            END case;

        when others =>
            Proximo_estado<=reset;
    end case;
end process;

end Behavioral;
```

1.3. TFG_wrapper.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library UNISIM;
use UNISIM.VCOMPONENTS.ALL;
entity TFG_wrapper is
  port (
    DDR_addr : inout STD_LOGIC_VECTOR ( 14 downto 0 );
    DDR_ba : inout STD_LOGIC_VECTOR ( 2 downto 0 );
    DDR_cas_n : inout STD_LOGIC;
    DDR_ck_n : inout STD_LOGIC;
    DDR_ck_p : inout STD_LOGIC;
    DDR_cke : inout STD_LOGIC;
    DDR_cs_n : inout STD_LOGIC;
    DDR_dm : inout STD_LOGIC_VECTOR ( 3 downto 0 );
    DDR_dq : inout STD_LOGIC_VECTOR ( 31 downto 0 );
    DDR_dqs_n : inout STD_LOGIC_VECTOR ( 3 downto 0 );
    DDR_dqs_p : inout STD_LOGIC_VECTOR ( 3 downto 0 );
    DDR_odt : inout STD_LOGIC;
    DDR_ras_n : inout STD_LOGIC;
    DDR_reset_n : inout STD_LOGIC;
    DDR_we_n : inout STD_LOGIC;
    FIXED_IO_ddr_vrn : inout STD_LOGIC;
    FIXED_IO_ddr_vrp : inout STD_LOGIC;
    FIXED_IO_mio : inout STD_LOGIC_VECTOR ( 53 downto 0 );
    FIXED_IO_ps_clk : inout STD_LOGIC;
    FIXED_IO_ps_porb : inout STD_LOGIC;
    FIXED_IO_ps_srstb : inout STD_LOGIC
  );
end TFG_wrapper;

architecture STRUCTURE of TFG_wrapper is
  component TFG is
  port (
    DDR_cas_n : inout STD_LOGIC;
    DDR_cke : inout STD_LOGIC;
    DDR_ck_n : inout STD_LOGIC;
    DDR_ck_p : inout STD_LOGIC;
    DDR_cs_n : inout STD_LOGIC;
    DDR_reset_n : inout STD_LOGIC;
    DDR_odt : inout STD_LOGIC;
    DDR_ras_n : inout STD_LOGIC;
    DDR_we_n : inout STD_LOGIC;
    DDR_ba : inout STD_LOGIC_VECTOR ( 2 downto 0 );
    DDR_addr : inout STD_LOGIC_VECTOR ( 14 downto 0 );
    DDR_dm : inout STD_LOGIC_VECTOR ( 3 downto 0 );
    DDR_dq : inout STD_LOGIC_VECTOR ( 31 downto 0 );
    DDR_dqs_n : inout STD_LOGIC_VECTOR ( 3 downto 0 );
    DDR_dqs_p : inout STD_LOGIC_VECTOR ( 3 downto 0 );
```

```
    FIXED_IO_mio : inout STD_LOGIC_VECTOR ( 53 downto 0 );
    FIXED_IO_ddr_vrn : inout STD_LOGIC;
    FIXED_IO_ddr_vrp : inout STD_LOGIC;
    FIXED_IO_ps_srstb : inout STD_LOGIC;
    FIXED_IO_ps_clk : inout STD_LOGIC;
    FIXED_IO_ps_porb : inout STD_LOGIC;
    icap_grant : in STD_LOGIC;
    monitor_rx : in STD_LOGIC;
    inject_strobe : in STD_LOGIC;
    reset_signal_status: in STD_LOGIC;
    clk : out STD_LOGIC;
    monitor_tx : out STD_LOGIC;
    status_observation : out STD_LOGIC;
    status_initialization : out STD_LOGIC;
    status_heartbeat : out STD_LOGIC;
    status_essential : out STD_LOGIC;
    status_injection : out STD_LOGIC;
    status_classification : out STD_LOGIC;
    status_correction : out STD_LOGIC;
    status_uncorrectable : out STD_LOGIC;
    GPIO_I : in STD_LOGIC_VECTOR ( 10 downto 0 );
    GPIO_O : out STD_LOGIC_VECTOR ( 10 downto 0 )
  );
  end component TFG;

--Circuito para el test
  component wrap_b13 is
      port(    clk: in std_logic;
               error: out std_logic );
  end component wrap_b13;


signal clk_parcial : STD_LOGIC;
signal error_parcial : STD_LOGIC;
signal monitor_rx_parcial : STD_LOGIC;
signal monitor_tx_parcial : STD_LOGIC;
signal GPIO_I_parcial : STD_LOGIC_VECTOR(10 downto 0);
signal GPIO_O_parcial : STD_LOGIC_VECTOR (10 downto 0);

begin

TFG_i: component TFG
    port map (
      DDR_addr(14 downto 0) => DDR_addr(14 downto 0),
      DDR_ba(2 downto 0) => DDR_ba(2 downto 0),
      DDR_cas_n => DDR_cas_n,
      DDR_ck_n => DDR_ck_n,
      DDR_ck_p => DDR_ck_p,
      DDR_cke => DDR_cke,
      DDR_cs_n => DDR_cs_n,
```

```
      DDR_dm(3 downto 0) => DDR_dm(3 downto 0),
      DDR_dq(31 downto 0) => DDR_dq(31 downto 0),
      DDR_dqs_n(3 downto 0) => DDR_dqs_n(3 downto 0),
      DDR_dqs_p(3 downto 0) => DDR_dqs_p(3 downto 0),
      DDR_odt => DDR_odt,
      DDR_ras_n => DDR_ras_n,
      DDR_reset_n => DDR_reset_n,
      DDR_we_n => DDR_we_n,
      FIXED_IO_ddr_vrn => FIXED_IO_ddr_vrn,
      FIXED_IO_ddr_vrp => FIXED_IO_ddr_vrp,
      FIXED_IO_mio(53 downto 0) => FIXED_IO_mio(53 downto 0),
      FIXED_IO_ps_clk => FIXED_IO_ps_clk,
      FIXED_IO_ps_porb => FIXED_IO_ps_porb,
      FIXED_IO_ps_srstb => FIXED_IO_ps_srstb,
      GPIO_I(10 downto 0) => GPIO_I_parcial(10 downto 0),
      GPIO_O(10 downto 0) => GPIO_O_parcial(10 downto 0),
      monitor_rx => monitor_rx_parcial,
      monitor_tx => monitor_tx_parcial,
      clk => clk_parcial,
      status_heartbeat => GPIO_I_parcial(0),
      status_initialization => GPIO_I_parcial(1),
      status_observation => GPIO_I_parcial(2),
      status_correction => GPIO_I_parcial(3),
      status_classification => GPIO_I_parcial(4),
      status_injection => GPIO_I_parcial(5),
      status_essential => GPIO_I_parcial(6),
      status_uncorrectable => GPIO_I_parcial(7),
      inject_strobe => GPIO_O_parcial(8),
      icap_grant => GPIO_O_parcial(9),
      reset_signal_status => GPIO_O_parcial(10)
    );

b_13: component wrap_b13
        port map(
            clk=> clk_parcial,
            error=> GPIO_I_parcial(11)
            );

end STRUCTURE;
```

2.Código en C del proyecto

2.1. main.c

```
#include <stdio.h>
#include "platform.h"
#include "xparameters.h"
#include "xil_io.h"
#include "xstatus.h"
#include "xdevcfg.h"
#include "xgpiops.h"
#include "xscugic.h"
#include "xil_exception.h"
#include "xscutimer.h"

#define SEM_BASEADDRESS          XPAR_SEM_IP_AXI_0_BASEADDR
#define DCFG_DEVICE_ID           XPAR_XDCFG_0_DEVICE_ID
#define GPIO_DEVICE_ID           XPAR_PS7_GPIO_0_DEVICE_ID
#define XSCUGIC_DEVICE_ID        XPAR_PS7_SCUGIC_0_DEVICE_ID
#define INTC_GPIO_INTERRUPT_ID   XPS_GPIO_INT_ID
#define XSCUTIMER_DEVICE_ID      XPAR_PS7_SCUTIMER_0_DEVICE_ID
#define XSCUTIMER_DEVICE_ID      XPAR_PS7_SCUTIMER_0_DEVICE_ID
#define INTC_TIMER_INTERRUPT_ID XPS_SCU_TMR_INT_ID

#define PCAP_PR_MASK      0x08000000   //PCAP_PR (bit 27 DEVCFG CTRL)
#define PCAP_MODE_MASK    0x04000000   //PCAP_MODE (bit 26 DEVCFG CTRL)

#define STATE_OBSERVATION_MASK   0x000A0000
#define STATE_IDLE_MASK          0x000E0000
#define INJECT_MASK              0x000C0000 //Tipo de inyeccion
                                             (Linear Frame Address)
#define BIT_32_MASK              0x80000000

#define INPUT    0
#define OUTPUT   1
#define Bank_2   XGPIOPS_BANK2
#define EDGE_SENSITIVE           0xFFFFFFFF
#define RISING_EDGE_GPIO         0xFFFFFFFF
#define SINGLE_EDGE              0x00000000
#define RISING_EDGE_GIC          0x03
#define PRIORITY_0               0
#define PRIORITY_8               8

XDcfg Dcfg;
XDcfg_Config *ConfigPtr_Dcfg;
XGpioPs  GPIO;
XGpioPs_Config  *ConfigPtr_GPIO;
XScuGic INTCInst;
XScuGic_Config *IntcConfig;
XScuTimer TIMER;
XScuTimer_Config *TimerConfig;
```

```
char gpio_activado[7]={0,0,0,0,0,0,0};
u8 Prescaler=255;
unsigned int Val_Timer1=100000;
unsigned int Val_Timer2=10000;
unsigned int i=0;

char error_circuito_test=0;
int error_incorregible=0;
int report=0;

/*Rutina de atencion a la interrupcion para el GPIO*/
void GPIO_Intr_Handler()
{
	XGpioPs_IntrDisable(&GPIO, Bank_2,0x000008BE);

	if(XGpioPs_ReadPin(&GPIO,55)==1)//initialization
	{
		gpio_activado[1]=1;
	}
	if(XGpioPs_ReadPin(&GPIO,56)==1)//observation
	{
		gpio_activado[2]=1;
	}
	if(XGpioPs_ReadPin(&GPIO,57)==1)//correction
	{
		gpio_activado[3]=1;
		XScuTimer_RestartTimer(&TIMER);
		XScuTimer_Stop(&TIMER);
	}
	if(XGpioPs_ReadPin(&GPIO,58)==1)//classification
	{
		gpio_activado[4]=1;
	}
	if(XGpioPs_ReadPin(&GPIO,59)==1)//injection
	{
		gpio_activado[5]=1;
	}
	if(XGpioPs_ReadPin(&GPIO,61)==1)//uncorrectable
	{
		gpio_activado[7]=1;
	}
	if(XGpioPs_ReadPin(&GPIO,65)==1)//Señal de error del circuito de test
	{
		error_circuito_test=1;
	}
	XGpioPs_IntrClear(&GPIO, Bank_2, 0x000008BE);
	XGpioPs_IntrEnable(&GPIO, Bank_2,0x000008BE);
}
```

```
/*Rutina de atención a la interrupción para el TIMER*/
void Timer_Intr_Handler()
{
	XScuTimer_DisableInterrupt(&TIMER);
	XScuTimer_RestartTimer(&TIMER);
	XScuTimer_Stop(&TIMER);
	if(i==5)
	{
		i=1;
	}
	if(i==4)
	{
		i=3;
	}
	if(i==6)
	{
		i=2;
	}
	report++;
	XScuTimer_ClearInterruptStatus(&TIMER);
	XScuTimer_EnableInterrupt(&TIMER);
}

int main()
{
	char impresion_errores=0;
	unsigned int cuenta_errores=0;
	unsigned int val_rand=0;
	u32 Inj_sup, Inj_inf;
	char error_corregido;
	unsigned int total_corregidos=0;
	unsigned int total_incorregibles=0;
	unsigned int error_test=0;

	/*Asignación de la semilla para el cálculo de las direcciones
	aleatorias*/
	printf("Escribe un valor para comenzar el cálculo aleatorio:\n\r");
	scanf("%d",&val_rand);
	printf("%d\n\r",val_rand);
	srand((unsigned) val_rand);

	/*SETUP*/
    init_platform();
    init_GPIO();
    init_TIMER();
    init_Dcfg();
	  while(1)
	  {
	  /*Proceso para la inyección de errores, en interacción con las
	  interrupciones*/
	  if(error_incorregible==0)
	  {
```

```
if(gpio_activado[1]==1)//status_initialization
{
	gpio_activado[1]=0;
}
if(gpio_activado[2]==1)//status_observation
{
	XScuTimer_LoadTimer(&TIMER, Val_Timer1);
	XScuTimer_Start(&TIMER);
	i=5;
	gpio_activado[2]=0;
}
if(gpio_activado[3]==1)//status_correction
{
	activar_reset_signal_status();
	gpio_activado[3]=0;
}
if(gpio_activado[4]==1)//status_classification
{
	activar_reset_signal_status();
	XScuTimer_LoadTimer(&TIMER, Val_Timer1);
	XScuTimer_Start(&TIMER);
	error_corregido=1;
	gpio_activado[4]=0;
}
if(gpio_activado[5]==1)//status_injection
{
	gpio_activado[5]=0;
}
if(gpio_activado[7]==1)//status_uncorrectable
{
	error_corregido=2;
	activar_reset_signal_status();
	gpio_activado[7]=0;
	report++;
	i=0;
}


/*Según el estado en que se halle, escribe una cosa u
  otra en el bus de inyección de errores*/
switch (i)
{
//Observation State
case 1:
	Paso_IDLE_STATE();
	i=6;
	XScuTimer_LoadTimer(&TIMER, Val_Timer2);
	XScuTimer_Start(&TIMER);
	break;
```

```
		//Idle State
		case 2:
			//Calculo las nuevas direcciones
			Inj_sup=Calculo_inject_address();
			Inj_inf=Calculo_inject_address();
			//Escribo las direcciones en el SEM
			Write_inject_address(Inj_sup, Inj_inf);
			cuenta_errores++;
			error_corregido=0;
			i=4;
			XScuTimer_LoadTimer(&TIMER, Val_Timer2);
			XScuTimer_Start(&TIMER);
			break;
		//Idle State después de inyectar error
		case 3:
			Paso_OBSERVATION_STATE();
			i=5;
			XScuTimer_LoadTimer(&TIMER, Val_Timer2);
			XScuTimer_Start(&TIMER);
			break;
		}
}

/*REPORT de los fallos inyectados*/
if(report!=0)
{
	report=0;
	if(impresion_errores==0)
	{
			if(error_corregido==1)
			{
				printf("Error %d: \n\rDireccion superior:
				%d\n\rDireccion inferior: %d\n\r"
				,cuenta_errores,Inj_sup,Inj_inf);
				printf("CORREGIDO\n\r");
			if(error_circuito_test==1)
			{
				error_circuito_test=0;
				printf("Este error pertenece al circuito en
				test\n\r");
				error_test++;
			}
			total_corregidos++;
			}
			if(error_corregido==2)
			{
				printf("Error %d: \n\r Dirección superior:
				%d\n\r Dirección inferior: %d\n\r"
					,cuenta_errores,Inj_sup,Inj_inf);
				printf("INCORREGIBLE\n\r");
```

```
                                if(error_circuito_test==1)
                                {
                                      error_circuito_test=0;
                                printf("Este error pertenece al circuito en
                                test\n\r");
                                      error_test++;
                                }
                                total_incorregibles++;
                                impresion_errores=1;
                                printf("\n\r Total errores
                                corregidos:%d\n\r",total_corregidos);
                                printf("Total errores incorregibles:%d\n\r",
                                total_incorregibles);
                                printf("Total errores en el circuito de
                                test:%d\n\r", error_test);
                                printf("Total errores
                                inyectados:%d\n\r",cuenta_errores);
                          printf("\n\rReinicie la FPGA para inyectar mas
                          fallos\n\r");
                    }
              }
        }
    }
    cleanup_platform();
    return 0;
}


/*Escribe el comando para pasar a estado Idle*/
void Paso_IDLE_STATE()
{
      Write_error_address(BIT_32_MASK |STATE_IDLE_MASK);
      Delay(20);
      activar_inject_strobe();
      Delay(20);
      desactivar_inject_strobe();
}

/*Escribe la direccion para inyectar el error*/
void Write_inject_address(u32 superior, u32 inferior)
{
      u32 superior_bis;
      superior_bis=(~INJECT_MASK & superior);
      superior_bis=(INJECT_MASK | superior_bis);
      Write_error_address(BIT_32_MASK |superior_bis);
      Delay(200);
      Write_error_address(~BIT_32_MASK & inferior);
      Delay(20);
      activar_inject_strobe();
      Delay(20);
      desactivar_inject_strobe();
}
```

```
/*Escribe el comando para pasar al estado de observacion*/
void Paso_OBSERVATION_STATE()
{
	Write_error_address(BIT_32_MASK |STATE_OBSERVATION_MASK);
	Delay(20);
	activar_inject_strobe();
	Delay(20);
	desactivar_inject_strobe();
}

/*Pone el dato en el bus AXI y lo dirige al controlador SEM*/
void Write_error_address(u32 valor)
{
	u32 offset=8;
	Xil_Out32((SEM_BASEADDRESS)+offset, valor);
}


/*Lee un dato del bus AXI proveniente del controlador SEM*/
int Read_error_address()
{
	u32 valor;
	u32 offset=8;
	valor=Xil_In32((SEM_BASEADDRESS)+offset);
	return valor;
}


/*Inicializa el Device Config Interface*/
void init_Dcfg()
{
	int Status;

	ConfigPtr_Dcfg = XDcfg_LookupConfig(DCFG_DEVICE_ID);
	Status = XDcfg_CfgInitialize(&Dcfg , ConfigPtr_Dcfg,
			ConfigPtr_Dcfg->BaseAddr);
	if(Status != XST_SUCCESS)
	{return XST_FAILURE;}

	/*Deshabilita PCAP_PR y activa PCAP_MODE, para permitir la
	inicializacion del SEM*/
	Enable_ICAP(&Dcfg);
	/*Activo la señal Icap_grant para el SEM*/
	XGpioPs_WritePin(&GPIO, 63, 1);
}
```

```
/*Habilita el acceso ICAP a la FPGA*/
void Enable_ICAP(XDcfg *InstancePtr)
{
	u32 CtrlReg;

	Xil_AssertVoid(InstancePtr != NULL);
	Xil_AssertVoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);

	CtrlReg = XDcfg_ReadReg(InstancePtr->Config.BaseAddr
					,XDCFG_CTRL_OFFSET);

	/*Escribe en el Registro de Control los valores para activar PCAP_MODE
	y desactivar PCAP_PR*/
	XDcfg_WriteReg(InstancePtr->Config.BaseAddr,
	XDCFG_CTRL_OFFSET,(CtrlReg | PCAP_MODE_MASK));
	XDcfg_WriteReg(InstancePtr->Config.BaseAddr,
	XDCFG_CTRL_OFFSET,(CtrlReg & ( ~PCAP_PR_MASK)));

}


/*Funcion de inicializacion del GPIO*/
void init_GPIO()
{
	int Status;

	ConfigPtr_GPIO=XGpioPs_LookupConfig(GPIO_DEVICE_ID);
	Status = XGpioPs_CfgInitialize(&GPIO,ConfigPtr_GPIO,
				ConfigPtr_GPIO->BaseAddr);
		if(Status != XST_SUCCESS)
		{return XST_FAILURE;}

	/*Configuro como entrada el  Bank_2 del GPIO (señales 54 a 85 del
	EMIO) excepto la 62, 63 y 64*/
	XGpioPs_SetDirection(&GPIO, Bank_2, INPUT);
	XGpioPs_SetDirectionPin(&GPIO, 62, OUTPUT);//inject_strobe
	XGpioPs_SetDirectionPin(&GPIO, 63, OUTPUT);//icap_grant
	XGpioPs_SetDirectionPin(&GPIO, 64, OUTPUT);//reset_signal_status
	XGpioPs_WritePin(&GPIO, 62, 0);
	XGpioPs_WritePin(&GPIO, 63, 0);
	XGpioPs_WritePin(&GPIO, 64, 0);
	Enable_interrupt_GPIO();
}
```

```
/*Habilita las interrupciones del GPIO habilitandola en el controlador de
interrupciones*/
void Enable_interrupt_GPIO()
{
	int status;

	IntcConfig = XScuGic_LookupConfig(XSCUGIC_DEVICE_ID);
	status = XScuGic_CfgInitialize(&INTCInst, IntcConfig,
					IntcConfig->CpuBaseAddress);
	if(status != XST_SUCCESS) return XST_FAILURE;

	Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
			(Xil_ExceptionHandler)XScuGic_InterruptHandler,
			&INTCInst);
		Xil_ExceptionEnable();

	status = XScuGic_Connect(&INTCInst,INTC_GPIO_INTERRUPT_ID,
			(Xil_ExceptionHandler)GPIO_Intr_Handler,(void *)&GPIO);
	if(status != XST_SUCCESS) return XST_FAILURE;

	XScuGic_SetPriorityTriggerType(&INTCInst,
	INTC_GPIO_INTERRUPT_ID,PRIORITY_0, RISING_EDGE_GIC);
	XGpioPs_SetIntrType(&GPIO, Bank_2, EDGE_SENSITIVE, RISING_EDGE_GPIO,
	SINGLE_EDGE);
	XGpioPs_IntrEnable(&GPIO, Bank_2,0x000008BE);
			//status_heartbeat => GPIO(0) no activada
			//status_essential => GPIO(6) no activada
	XScuGic_Enable(&INTCInst, INTC_GPIO_INTERRUPT_ID);
}


/*Inicializa el TIMER*/
void init_TIMER()
{
	int status;

	TimerConfig=XScuTimer_LookupConfig(XSCUTIMER_DEVICE_ID);
	status= XScuTimer_CfgInitialize(&TIMER, TimerConfig,
				TimerConfig->BaseAddr);
	if(status != XST_SUCCESS) return XST_FAILURE;

	Enable_interrupt_TIMER();
	//Configuramos el timer para 20ms
	XScuTimer_SetPrescaler(&TIMER, Prescaler);
	XScuTimer_LoadTimer(&TIMER, Val_Timer2);
}
```

```
/*Habilita las interrupciones del TIMER y las conecta con el GIC*/
void Enable_interrupt_TIMER()
{
	int Status;

	Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_IRQ_INT,

	(Xil_ExceptionHandler)XScuGic_InterruptHandler,&INTCInst);
	Xil_ExceptionEnable();

	Status = XScuGic_Connect(&INTCInst, XPS_SCU_TMR_INT_ID,
			(Xil_ExceptionHandler)Timer_Intr_Handler,(void *)&TIMER);
	if (Status != XST_SUCCESS) {return Status;}

	XScuGic_SetPriorityTriggerType(&INTCInst,
	INTC_TIMER_INTERRUPT_ID,PRIORITY_8, RISING_EDGE_GIC);

	XScuTimer_EnableInterrupt(&TIMER);

	XScuGic_Enable(&INTCInst, INTC_TIMER_INTERRUPT_ID);
}

void activar_inject_strobe()
{
	XGpioPs_WritePin(&GPIO, 62, 1);
}

void desactivar_inject_strobe()
{
	XGpioPs_WritePin(&GPIO, 62, 0);
}

void activar_reset_signal_status()
{
	XGpioPs_WritePin(&GPIO, 64, 1);
	Delay(20);
	XGpioPs_WritePin(&GPIO, 64, 0);
}


void Delay(int i)
{
	int j;
	for(j=0;j<=i;j++){}
}

/*Calcula aleatoriamente la dirección de error*/
int Calculo_inject_address()
{
	u32 inject_address;

	inject_address=rand();
	return inject_address;
}
```

Bibliografía

[1] L. H. Crockett, R. A. Elliot, M. A. Enderwitz and R. W. Stewart, *The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*, First Edition, Strathclyde Academic Media, 2014.

[2] Zynq-7000 All Programmable SoC Technical Reference Manual, Xilinx, v1.10, February 2015.

[3] Soft Error Mitigation Controller LogiCORE IP Product Guide, Xilinx, v4.1, November 2014.

[4] Rich Griffin, Designing a Custom AXI-lite Slave Peripheral, Version 1.0, Silica EMEA, July 2014.

[5] Vivado Design Suite User Guide, Xilinx, v2014.4, November 2014.

[6] A. Rodríguez Pérez, Estudio y ensayo de un circuito digital para una prueba en radiación ionizante, Universidad de Sevilla (PFC), Septiembre 2007.

[7] D. Sánchez Gallego, Desarrollo multiplataforma de un PSoC basado en microprocesador ARM para aplicaciones empotradas, Universidad Politécnica de Cartagena (PFC), Septiembre 2014.

[8] S. Ortega Lázaro, Implementación de periféricos en Vivado para dispositivos Zynq, Universidad de Alcalá (TFG), 2014.

[9] Wikipedia, Available: https://www.wikipedia.org/ [Ultimo acceso: Julio 2015].

[10] Xilinx Forums, Available: http://forums.xilinx.com/ [Ultimo acceso: Junio 2015].

[11] ITC'99 Benchmarks (2nd release). http://www.cad.polito.it/downloads/tools/itc99.html